US006154738A

United States Patent [19]

Call

[11] Patent Number: 6,154,738

[45] Date of Patent: *Nov. 28, 2000

[54] METHODS AND APPARATUS FOR DISSEMINATING PRODUCT INFORMATION VIA THE INTERNET USING UNIVERSAL PRODUCT CODES

[76] Inventor: Charles Gainor Call, 53 Saint Stephen St., Boston, Mass. 02115

[*] Notice: This patent is subject to a terminal disclaimer.

[21] Appl. No.: 09/316,597

[22] Filed: May 21, 1999

Related U.S. Application Data

[63] Continuation-in-part of application No. 09/049,426, Mar. 27, 1998, Pat. No. 5,913,210.

[51] Int. Cl.[7] G06F 15/173

[52] U.S. Cl. 707/4; 707/513; 707/3; 707/10; 705/20; 705/23; 705/26; 709/201; 709/213; 709/217; 709/249

[58] Field of Search 707/10, 4, 3, 513, 707/524; 705/20, 23, 26; 395/200.31; 709/201, 213, 217, 249

[56] References Cited

U.S. PATENT DOCUMENTS

5,794,221	8/1998	Egendorf	705/40
5,804,803	9/1998	Cragun et al.	235/375
5,918,214	6/1999	Perkowski	705/27
5,950,173	9/1999	Perkowski	705/26
6,045,048	4/2000	Wilz, Sr. et al.	235/472.01

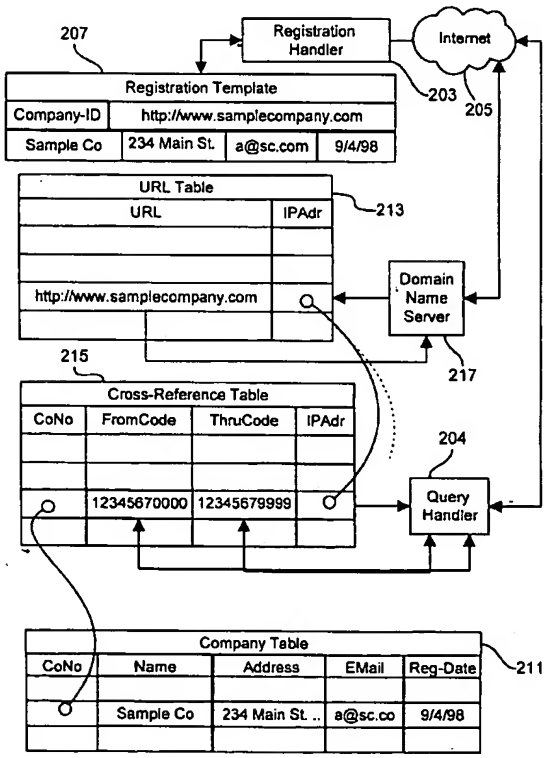
Primary Examiner—Jean R. Homere

[57] ABSTRACT

Methods and apparatus for disseminating over the Internet product information produced and maintained by product manufacturers using existing universal product codes (bar codes) as access keys. A cross-referencing resource, which may take the form of an independent HTTP server, an LDAP directory server, or the existing Internet Domain Name Service (DNS), receives Internet request messages containing all or part of a universal product code and returns the Internet address at which information about the identified product, or the manufacturer of that product, may be obtained. By using preferred Web data storage formats which conform to XML, XLS, XLink, Xpointer and RDF specifications, product information may be seamlessly integrated with information from other sources. A "web register" module can be employed to provide an Internet interface between a shared sales Internet server and an otherwise conventional inventory control system, and operates in conjunction with the cross-referencing server to provide detailed product information to Internet shoppers who may purchase goods from existing stores via the Internet.

14 Claims, 8 Drawing Sheets

Microfiche Appendix Included
(1 Microfiche, 28 Pages)



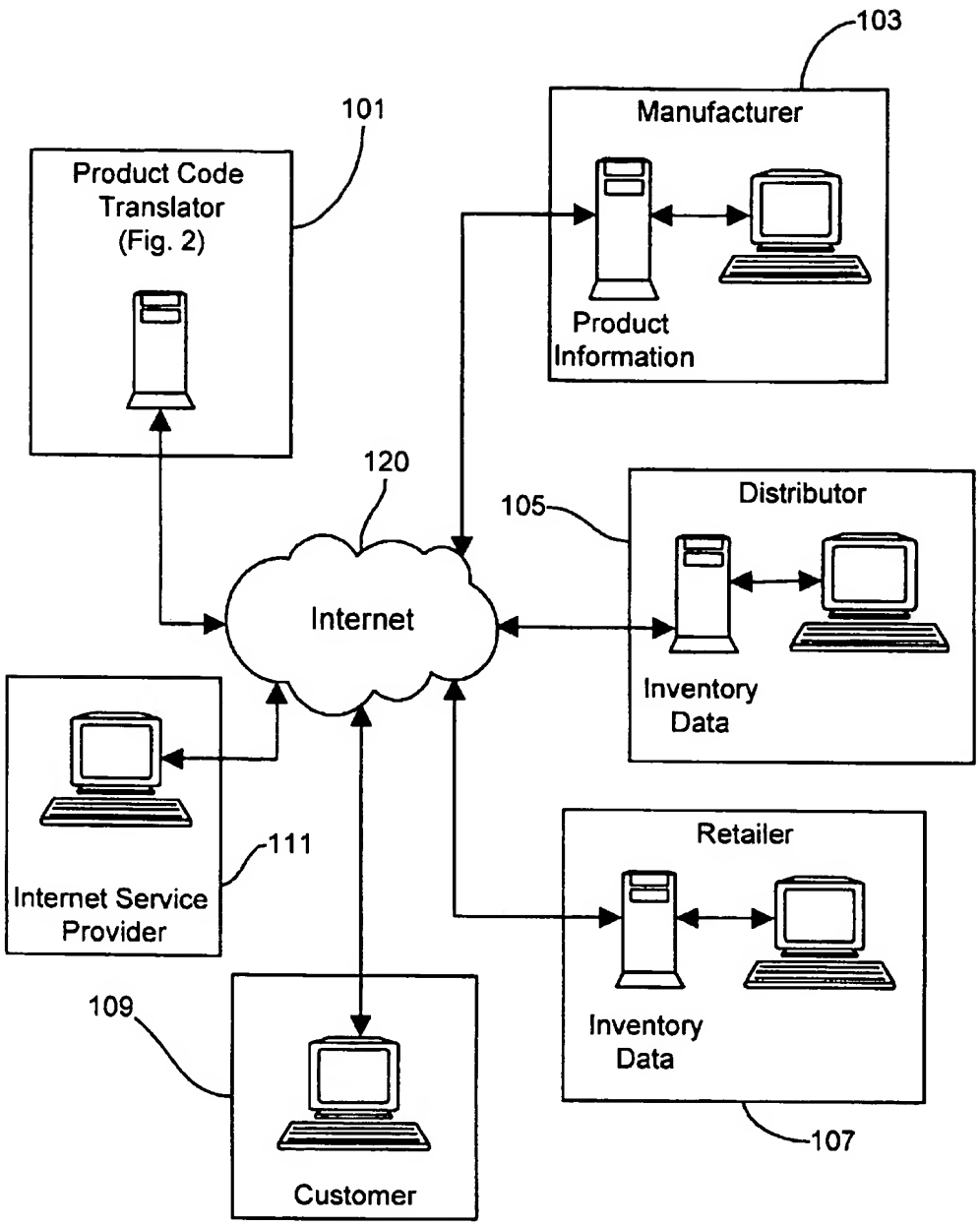

103
Manufacturer
101
Product Code Translator (Fig. 2)
Product Information
120
105
Distributor
Internet
Inventory Data
111
Internet Service Provider
Retailer
109
Inventory Data
Customer
107

Fig. 1

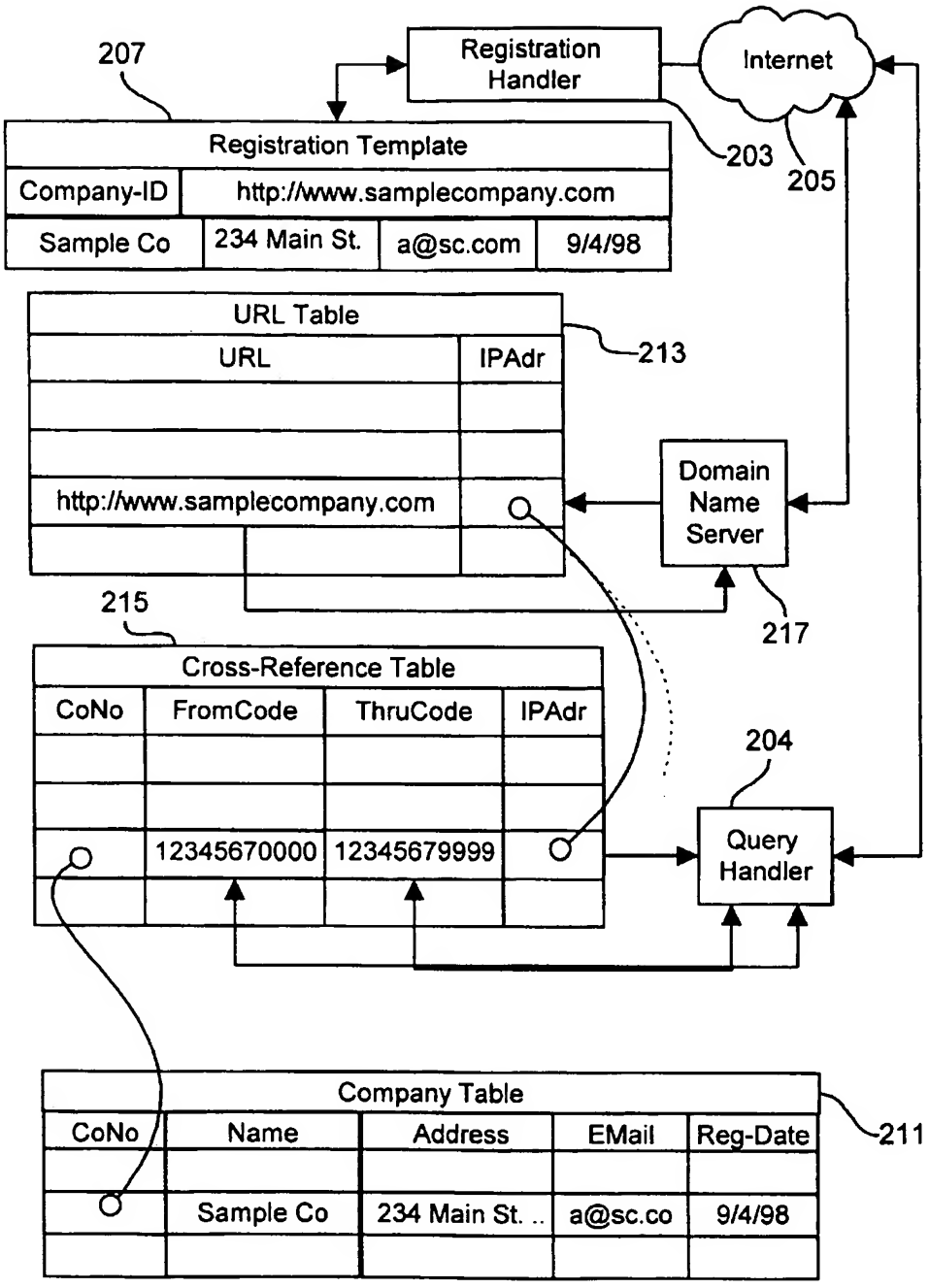

207
Registration Handler
Internet
203
205
Registration Template
Company-ID
http://www.samplecompany.com
Sample Co
234 Main St.
a@sc.com
9/4/98
URL Table
213
URL
IPAdr
http://www.samplecompany.com
Domain Name Server
217
215
Cross-Reference Table
CoNo
FromCode
ThruCode
IPAdr
12345670000
12345679999
204
Query Handler
Company Table
211
CoNo
Name
Address
EMail
Reg-Date
Sample Co
234 Main St. ..
a@sc.co
9/4/98

Fig. 2

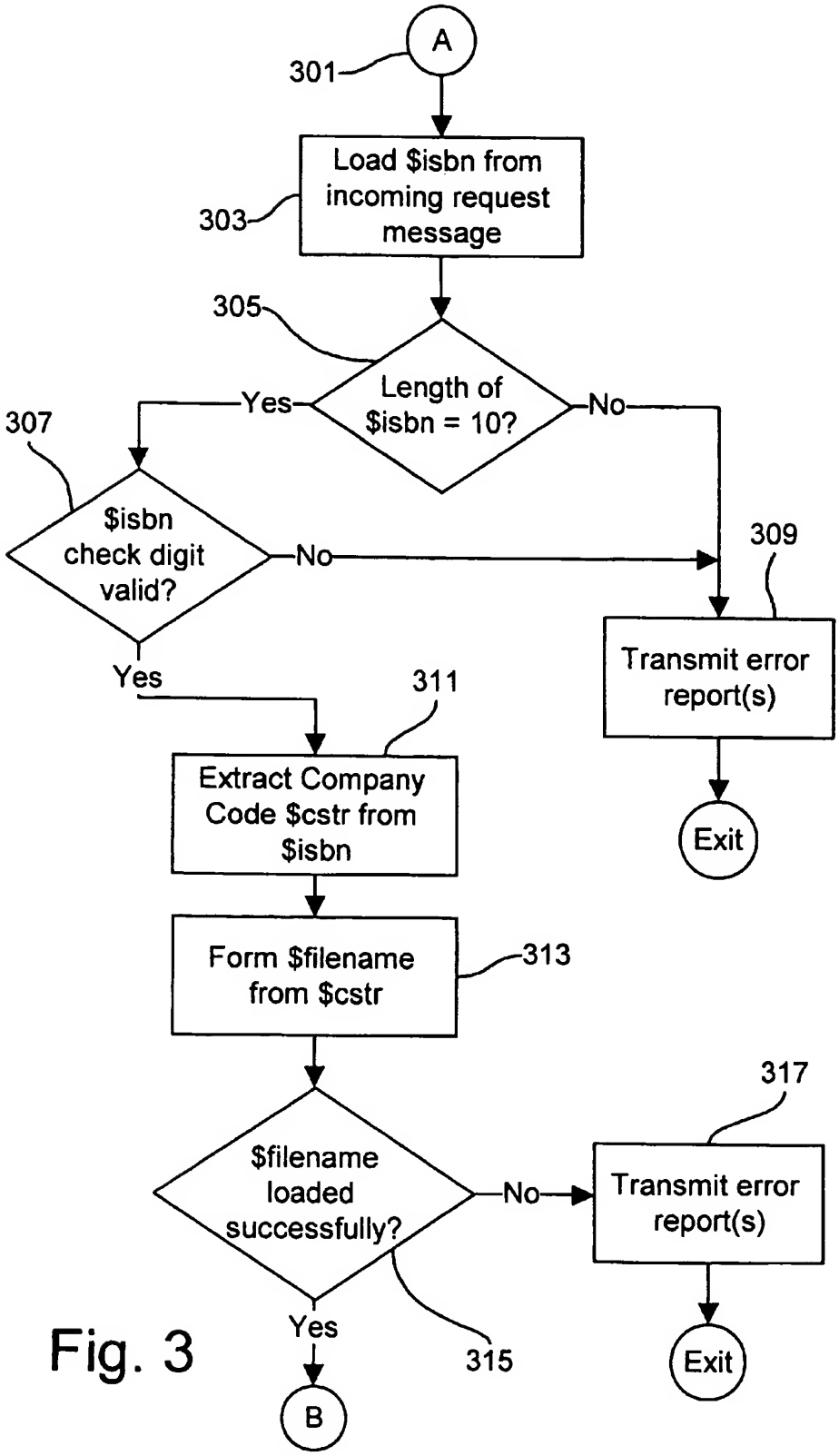

A
301
Load $isbn from incoming request message
303
305
Length of $isbn = 10?
Yes
No
307
$isbn check digit valid?
No
309
Transmit error report(s)
Exit
Yes
311
Extract Company Code $cstr from $isbn
Form $filename from $cstr
313
317
$filename loaded successfully?
No
Transmit error report(s)
Exit
Yes
315
B

Fig. 3

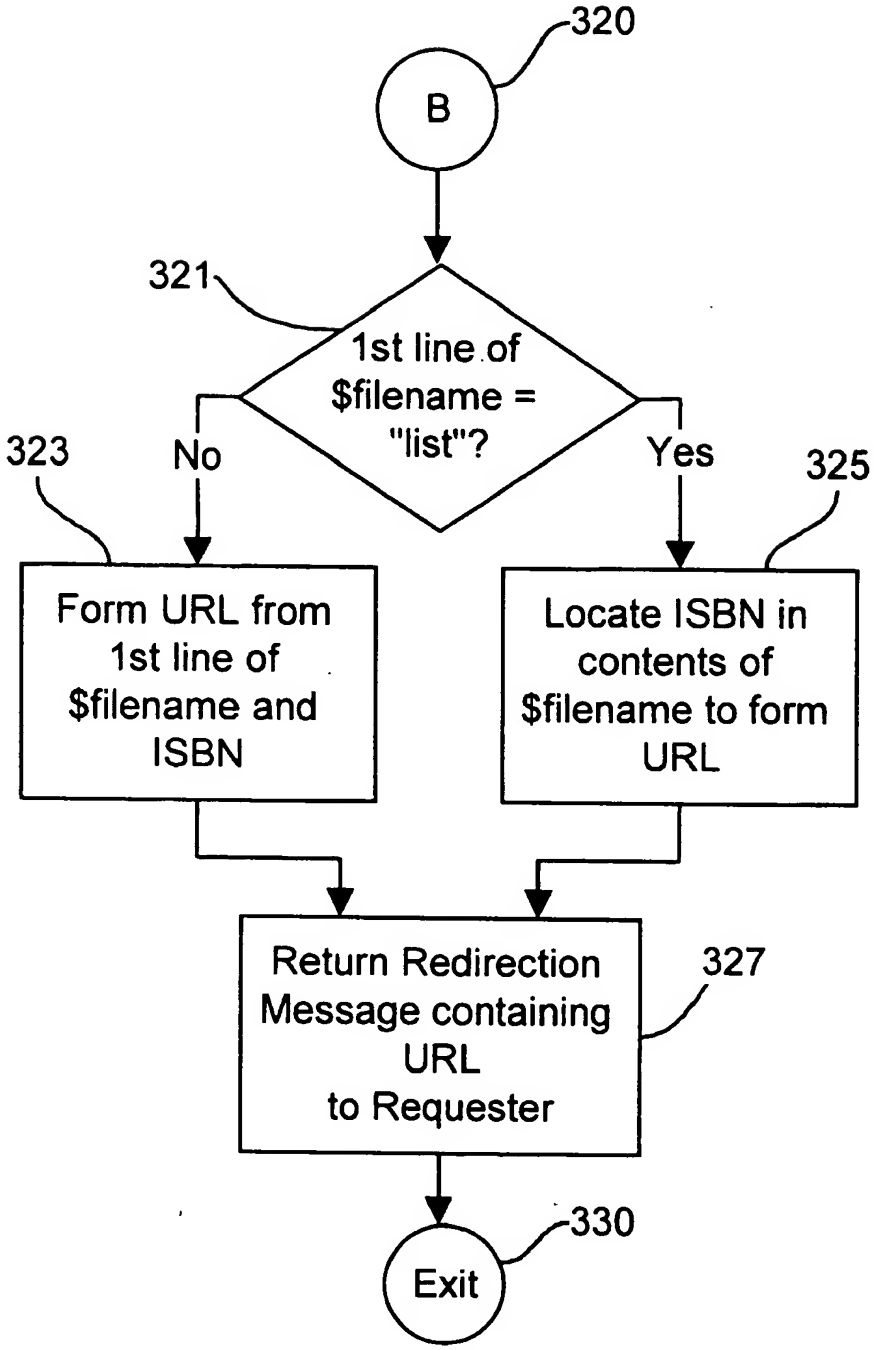

320
B
321
1st line of $filename = "list"?
No
Yes
323
325
Form URL from 1st line of $filename and ISBN
Locate ISBN in contents of $filename to form URL
Return Redirection Message containing URL to Requester
327
330
Exit

Fig. 4

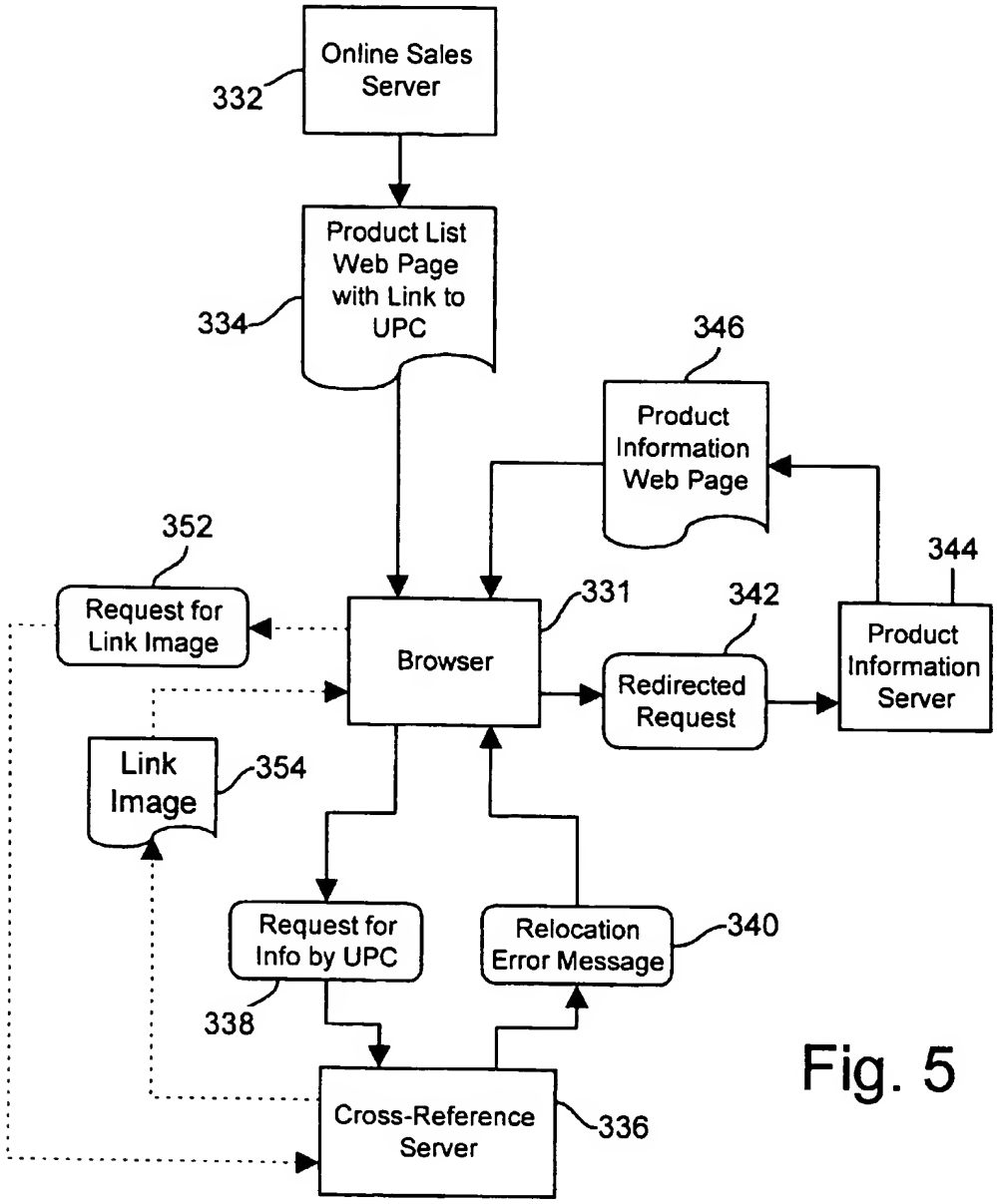

Online Sales Server
332
Product List Web Page with Link to UPC
334
346
Product Information Web Page
352
Request for Link Image
Browser
331
342
Redirected Request
344
Product Information Server
Link Image
354
Request for Info by UPC
338
Relocation Error Message
340
Cross-Reference Server
336

Fig. 5

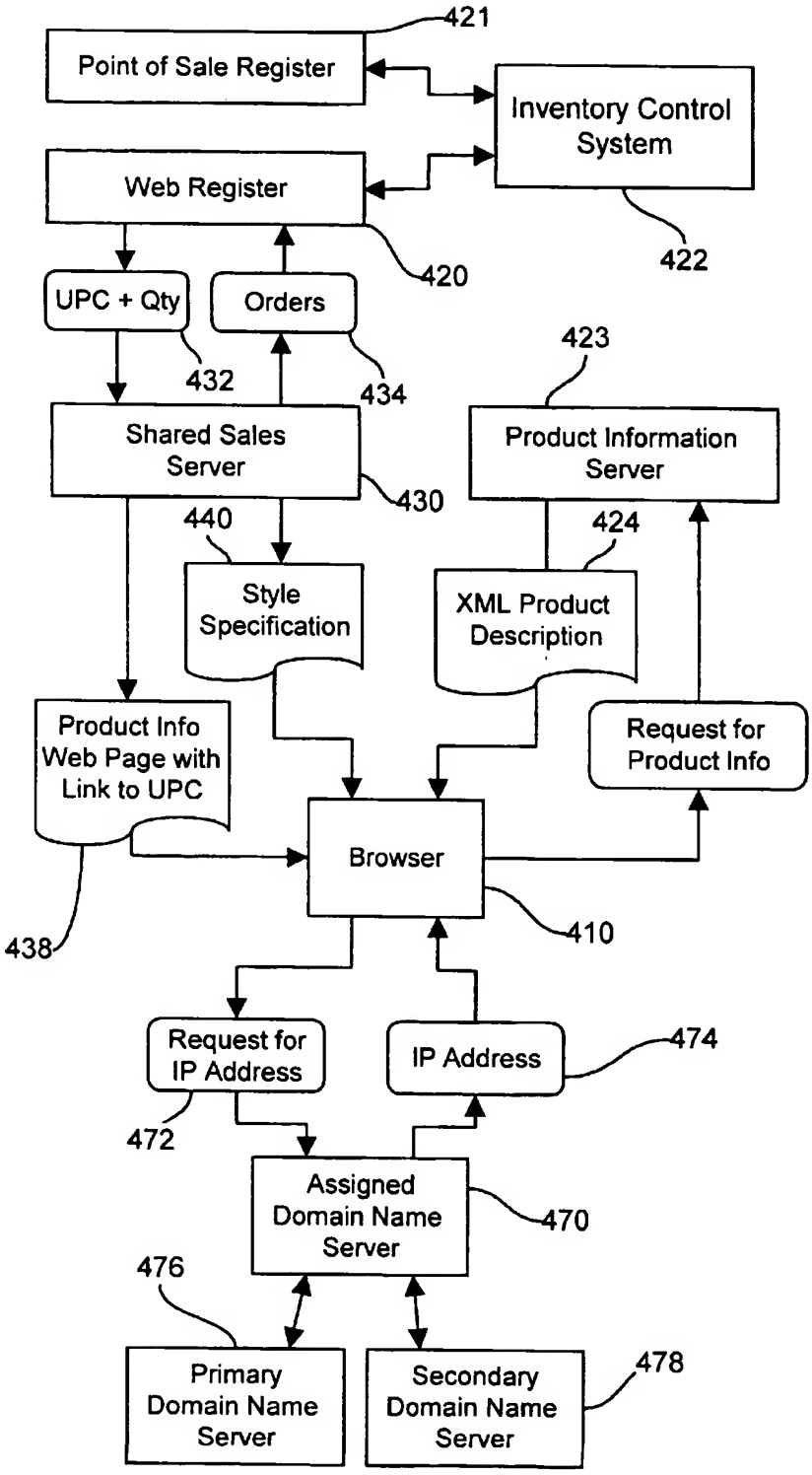

421
Point of Sale Register
Inventory Control System
Web Register
422
420
UPC + Qty
Orders
432
434
423
Shared Sales Server
Product Information Server
430
440
424
Style Specification
XML Product Description
Product Info Web Page with Link to UPC
Request for Product Info
Browser
438
410
Request for IP Address
IP Address
474
472
Assigned Domain Name Server
470
476
Primary Domain Name Server
Secondary Domain Name Server
478

Fig. 6

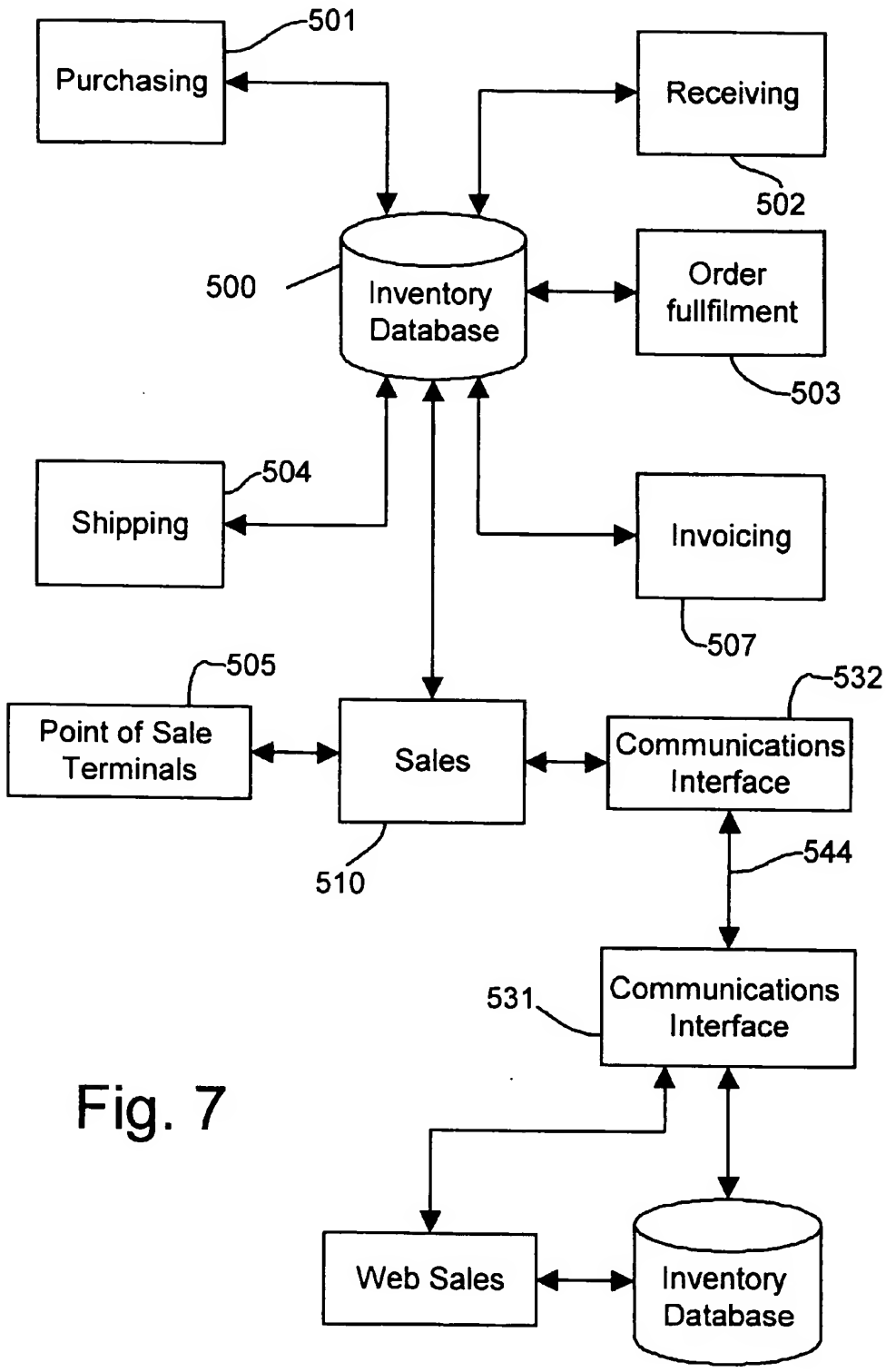

501
Purchasing
Receiving
502
500
Inventory Database
Order fullfilment
503
504
Shipping
Invoicing
507
505
Point of Sale Terminals
Sales
532
Communications Interface
544
510
531
Communications Interface
Web Sales
Inventory Database

Fig. 7

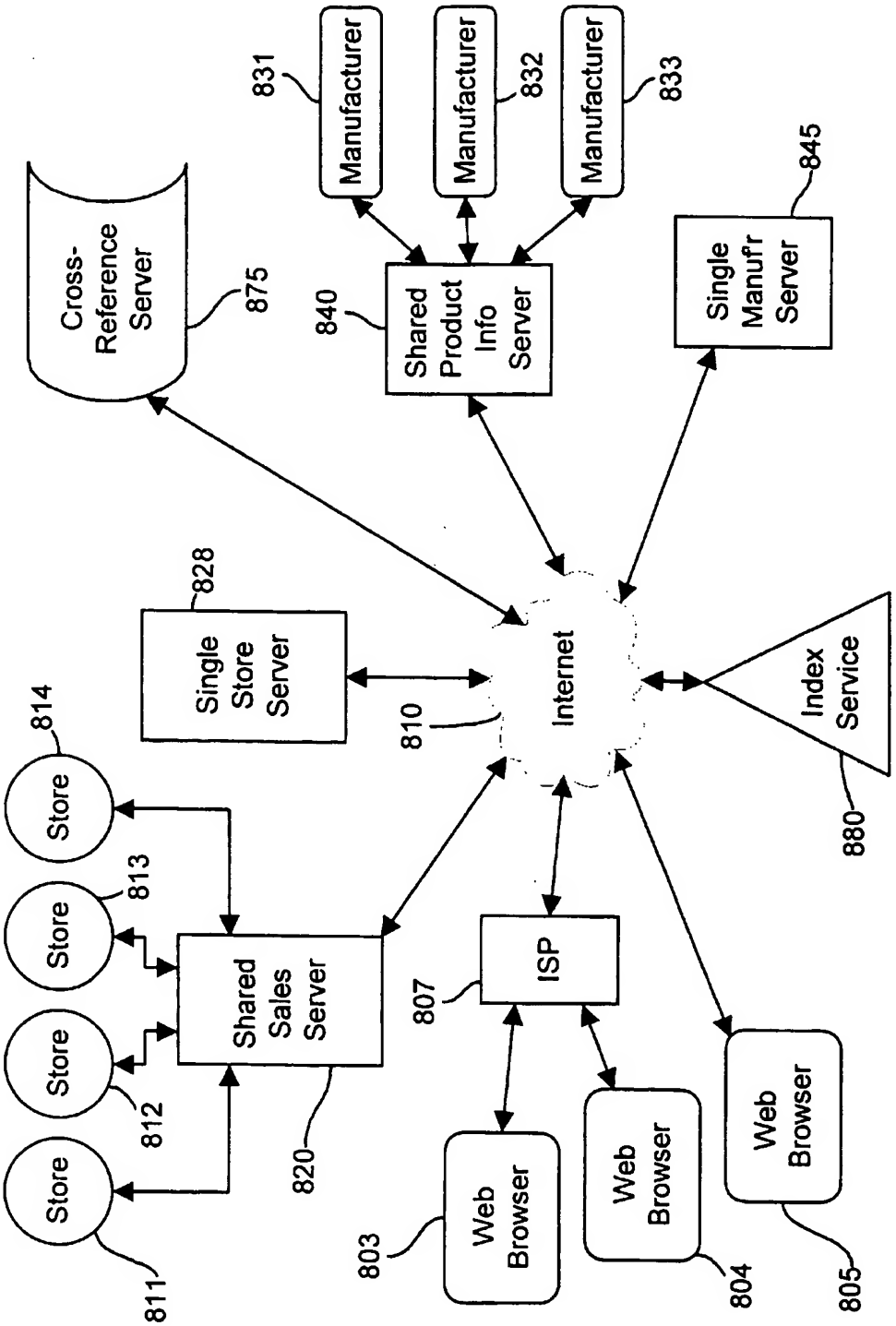

Manufacturer
831
Manufacturer
832
Manufacturer
833
Cross-Reference Server
875
840
Shared Product Info Server
Single Manufr Server
845
Single Store Server
828
Internet
810
Index Service
880
Store
814
Store
813
Store
812
Store
811
Shared Sales Server
820
ISP
807
Web Browser
803
Web Browser
804
Web Browser
805

Fig. 8

METHODS AND APPARATUS FOR DISSEMINATING PRODUCT INFORMATION VIA THE INTERNET USING UNIVERSAL PRODUCT CODES

CROSS-REFERENCE TO RELATED APPLICATIONS

This is a continuation in part of U.S. application Ser. No. 09/049,426 filed on Mar. 27, 1998 and entitled "Methods and Apparatus for Disseminating Product Information via the Internet", now U.S. Pat. No. 5,913,210.

REFERENCE TO MICROFICHE APPENDIX

A microfiche appendix consisting of 28 frames on one microfiche accompanies this specification and contains Perl language CGI scripts (computer source language listings) which illustrate working illustrative embodiments of selected components of the invention. A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

FIELD OF THE INVENTION

This invention relates to methods and apparatus for transferring requests for specific information to preferred sources of that information on the Internet.

BACKGROUND AND SUMMARY OF THE INVENTION

Manufacturers must provide information about their products to resellers, consumers, and others. Resellers need product information to select, promote and support the products they distribute. Consumers need information about available products to make informed buying choices. Advertisers, product analysts, manufacturer's representatives, shippers, and others also need information about the goods with which they deal.

Under current practices, product information typically originates with manufacturers and is primarily distributed in conventional print media advertising and product packaging. This information is often incomplete, difficult to update, and available only to a limited distribution. While the advent of the World Wide Web has permitted manufacturers to make detailed, up-to-date product information available via the Internet, the information describing a specific product is often difficult to locate, particularly when the URL (uniform resource locator) of the manufacturer's website is not known.

It is a general object of the present invention to transfer a request for information specified by an identifier, such as a product code, to a preferred source of that information, such as an Internet information resource devoted to the product specified by the product code which is created and maintained by the product's manufacturer.

The preferred embodiment of the present invention employs an Internet resource, called a "product code translator," for storing cross-references between universal product codes identifying specific products and Internet addresses specifying the locations at which information about these products may be obtained. The cross-references specify the universal product codes assigned to the participating manufacturers, such as the U.P.C. and EAN codes

widely used in retail stores for barcode scanning at checkout counters, and the Internet addresses where information can be obtained about the products designated by those codes.

In a principle aspect, the present invention takes the form of methods and apparatus for delivering information about products and manufacturers via the Internet using all or part of the universal product codes which designate these products and manufacturers as Internet access keys. The product information is stored in Internet servers, preferably in XML format, by the manufacturers who both produce the products and control the content of the stored product information.

Internet shoppers and others who desire product information fetch web pages via the Internet from on-line merchants and other sources. These web pages may contain one or more links to product information, and each such link contains a reference which designates a particular product by its corresponding universal product code. When the web browser operated by the shopper activates such a link, a request message containing at least a portion of the universal product code is sent via the Internet to a cross-referencing database, preferably maintained by the Internet Domain Name Service, which returns the Internet address of the particular manufacturer's server which then makes the desired product information available.

The present invention may be used to particular advantage to provide product information to web customers who visit web sites operated on behalf of retail stores which use universal product codes both for bar code checkout and to identify specific products in a computerized inventory control system. The retailer's inventory control systems need not store detailed product information since, by means of the invention, the universal product codes of items being offered for sale can be used to access product information directly from the manufacturer's servers.

In one embodiment of the invention, the cross-referencing function may be performed by a server which receives a hypertext transport protocol (HTTP) request message containing a universal product code, performs a lookup operation using a stored database of cross-references, and returns an HTTP response message which includes a location header field containing a destination URL specifying said particular Internet address. The requesting web browser then automatically redirects the request message to the destination URL.

The company code portion only of the universal product code may be stored in the cross-referencing database to refer a product information inquiry to the server operated by the manufacturer, with the remainder of the product code being sent to the manufacturer's server to identify the particular product. This reduces the size of the cross-referencing database, and further simplifies the process of registering manufacturers and maintaining the database.

When the cross-referencing server takes the form of an independently operated server, the standard Light-weight Directory Access Protocol can be employed to advantage to provide cross-references between all or part of each universal product code and the Internet address where information about that product may be obtained. Access to product information can be even more simplified and expedited by using the existing Internet Domain Name service to perform cross-referencing, a capability which can be added to the existing Internet infrastructure by simply reserving a preassigned name space for product code to address conversion.

By storing product information expressed in eXtensible Markup Language (XML), and by using stylesheet information provided by the web site which is incorporating product information into their web presentations, the data

supplied by the manufacturer can be rendered using font sizes, typefaces, background colors and formatting selected by the web page producer. Other characteristics of XML, including the ability to encourage or enforce conformity with content and formatting standards through the use of Document Type Definitions (DTD's)and the Resource Definition Framework (RDF) and Syntax Specification, facilitate the integration of data from retailers and other web page producers with the product information provided by manufacturers.

When the manufacturers of some products identified on web pages have not made product descriptions available, it is desirable to suppress the creation of visible link anchors which encourage users to attempt to activate links which will not work as intended. Such link suppression can be accomplished by determining the status of each desired link before the web page containing the product list is displayed. "Image-cued links" perform this function by employing, as link anchors, visible web page components such as image files which are retrieved via the cross-referencing server at the time the web page is displayed.

These and other objects, features and advantages of the present invention will be made more apparent through a consideration of the following detailed description of a preferred embodiment of the invention. In the course of this description, frequent reference will be made to the attached drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a schematic block diagram illustrating various instrumentalities which make use of the invention interconnected via the Internet;

FIG. **2** is a diagram illustrating the interrelationship of the principle data structures used to implement a product code translator of the type contemplated by the invention;

FIGS. **3** and **4** are a flow diagram depicting the operation of a CGI program which implements the product code translator, responding to request HTTP messages containing universal product codes and redirecting those request messages to the URLs where information about designated products may be found;

FIG. **5** is a data flow diagram illustrating the manner in which a web browser interacts with a server which acts as a product code translator by redirecting links on a merchant's web page to product information made available by the manufacturer;

FIG. **6** is a data flow diagram illustrating a further embodiment of the invention in which a web browser interacts with a domain name server which serves as a product code translator, redirecting links on a merchant's web page to XML data provided by a shared sales server used by several merchants, the XML data from the manufacturer being displayed in accordance with an XSL stylesheet specification unique to each merchant, and the online merchant functions being implemented by the shared sales server which is connected to the conventional inventory control system operated by each of the merchants; and

FIGS. **7** and **8** are a block diagrams illustrating the principle components of a typical retailer's inventory control system and the interface between that system and a shared sales server which maintains a parallel but more limited inventory database used during on-line sales transactions.

DETAILED DESCRIPTION

The present invention takes advantage of two existing and highly successful technologies: the Internet and the universal product code system. In accordance with the present invention, the Internet is employed to provide low cost, worldwide, bi-directional communication which enables product information to be requested by and sent to any person or firm using one or more universal product codes as information access keys. The universal product code system is in widespread use to uniquely identify each of the thousands of different suppliers and millions of different items that are warehoused, sold, delivered and billed throughout commercial channels of distribution. In accordance with the invention, a product code translator which may be accessed via the Internet provides information enabling the translation of universal product codes into the associated Internet addresses at which information about the products specified by those product codes may be obtained.

The present invention enables the retrieval of information about products from the source of those products, typically the manufacturer, by those who need that information, such as resellers and consumers. In accordance with the invention, any person or firm having access to the Internet and knowing the universal product code for a product may obtain information about that product from the participating manufacturer which supplies that product. The system employs a product code translator, which may be implemented by a plurality of servers but which is illustrated by the single resource seen at **101** in FIG. **1**. The product code translator is accessed via the Internet to perform a translation of specified universal product codes into the corresponding Internet addresses from which information about the designated products can be obtained.

The product code translator **101** stores cross-references between product codes and Internet addresses. The product codes and the Internet addresses are provided by or on behalf of participating manufacturers and suppliers, such as the manufacturer illustrated at **103** in FIG. **1**. These cross-references may then be retrieved from the cross-reference resource **101** by resellers, prospective buyers, as illustrated by the distributor **105**, the retailer **107** and the customer **109** seen in FIG. **1**. Internet Service Providers, as illustrated by the ISP **111** in FIG. **1**, may also utilize the data provided by the product code translator **101** to provide a variety of services and functions.

Before further describing how these entities function within the system, it will be useful to clarify some of the terms which will be used in this specification:

The term "universal product codes" (lower case) is used to indicate standardized industry or inter-industry codes used to designate items, packages and services made, used, leased or sold in commerce. The term thus includes the Universal Product Codes ("U.P.C.s") used by suppliers in the United States and Canada and managed by the Uniform Code Council, Inc., 8163 Old Yankee Road, Dayton, Ohio 45458; the EAN codes used by suppliers outside the U.S. and Canada under the general direction of EAN International, rue Royale 145, 1000 Bruxelles—Belgium; and any other multi-industry or single industry standard product designation system.

The term "manufacturer" will be used to refer to manufacturers, suppliers, vendors, licensors and others to whom sets of universal product codes have been assigned, or their agents. Typically, this assignment takes the form of the designation of a particular value for a portion of the universal product code which is reserved for exclusive use by a particular manufacturer. For example, the entity to which a specific six-digit "company-identifier" portion of a 12-digit numeric U.P.C. code has been assigned is a "manufacturer" as that term is used in this specification.

The term "product" is used to refer to a kind of item which is uniquely identified by a single universal product code, as opposed to a specific individual item of that kind. For example, a specific U.P.C. code is assigned by the manufacturer, Hershey Foods Corp., to 'Reese's Creamy Peanut Butter" as packaged in 510 gram containers (a "product") whereas a different U.P.C. code is assigned to the same peanut butter packaged in containers of a different size (a different "product").

The term "Internet address" will be used to refer to the all, or a significant part of, a reference to a resource on the Internet. Such a reference may take the form of a numerical IP address or an alphanumeric Uniform Resource Locator ("URL") which may identify a file on a specified machine, a database query, a specific command output, or some other accessible Internet resource. Thus, the term "Internet address" includes such things as a specific 32-bit address of a specific computer connected to the Internet, written in decimal as "123.040.212.002"; a domain name such as "patentsoft.com" which can be resolved into a numerical IP-address using a domain name server; the URL of a file accessible via the Internet, such as "ftp://www.sample.com/director/filename.xxx"; a URL identifying a query processing script with passed parameters, such as "http://xxx.yyyy.com/cgi/search%01234567890123"; or an email address such as "847563@manufacturer.com".

The Product Code Translator

The product code translator seen at **101** in FIG. **1** performs two primary functions illustrated in FIG. **2**: (1) its registration handler **203** accepts cross-references submitted by manufacturers which relate their assigned universal product codes to associated Internet addresses where information relating to their products may be obtained, and (2) its query handler **204** accepts queries via the Internet **205**, each query including all or part of one or more universal product codes, and returns the Internet addresses which can be used to obtain information about the products identified by those codes. The product code translator **101** may also advantageously perform other functions, examples of which are described below.

As seen in FIG. **1**, the product code translator **101** may be advantageously implemented by a server computer which stores information in a relational database consisting of the tables depicted in FIG. **2**. If desired, the product code translator **101** may be implemented with a plurality of "mirrored" servers at different locations, or clustered servers at the same location containing the same cross-referencing data to share the processing burden and provide redundant fault-tolerant reliability. In addition, different servers or sets of mirrored or clustered servers may be used to process different assigned subsets of the gamut of universal product codes. Whether one or many servers is used, each may be preferably implemented using conventional server hardware and conventional server operating system software, such as Microsoft NT Server, Netscape Application Server, SCO Unixware, Sun Enterprise Server, and the like.

The registration handler process **203**, which may be implemented on a server which stores cross-references, or on a separate server operated by a central registration authority, receives each registration submission via the Internet **205** to create an incoming registration data illustrated by the data template record **207**. The registration template record **207** includes several fields: an Company-ID field which holds the company-identifier portion of a universal product code in EAN format; a URL field which holds the Uniform Record Locator constituting the "base address" at which information can be retrieved about products designated by those universal product codes, company information fields which include the company's name, mailing address and email address fields so that the manufacturer submitting the registration can be identified and contacted; and a date field which specifies the date upon which the registration was first made.

The registration handler **203** may obtain the submitted data needed to create the registration template record in a variety of ways, such as accepting a HTML web page form completed and submitted via the Internet by a registrant, processing an incoming email message containing the necessary information, or receiving the needed information by telephone or regular mail.

The registration handler process preferably incorporates a mechanism or procedure for insuring that the registrant has the authority to create and alter the information being supplied. A variety of methods for enhancing the security of the registration process may be employed, including the issuance of a password at the time a range of universal product codes is first registered, with the requirement that the same password be thereafter provided by anyone who seeks to alter the information originally provided with respect to any product code within that previously registered range. The registration procedure may also require each registrant to provide an email contact address to which an email message of predetermined content is sent after the initial registration form is completed, to which the registrant must respond within a predetermined time to verify the registration. Any attempt thereafter to change the contact email address results in a message being sent to the originally registered email contact address advising that an attempt is being made to alter the registration. Finally, email confirmation may be requested from the email address registered with InterNIC for the URL to which universal product codes are to be linked. This step confirms that the person attempting a registration in fact has authority to link to that host computer and provides an additional safeguard against unauthorized submissions.

Instead of maintaining a user name and password database, the registration handler can use a database of certificates, such as Certificate Server available from Netscape, to create, sign, and manage certificates for all participating manufacturers, configuring other servers to accept only authorized user certificates. A scalable database may be used to store the status of each certificate, and the issuance and revocation of certificates can be centrally administered from the product code translator or a separate registration authority. Similar password, certificate or digital signature protection schemes may be used to provide access to certain data or to data in certain forms only to authorized requesters.

The information contained in the incoming registration template **207** is used to create records (rows) in three separate tables in the relational database: a company table **211**, a URL table **213** and a cross-reference table **215**. As seen in FIG. **2**, the company table **211** includes a numerical company number field CoNo which is also present in the cross-reference table **215** so that each cross-reference table row can be related to a particular company description record in the company table which has the same CoNo value. The key field CoNo establishes a one-to-many relationship between the company table **211** and the cross-reference table **215** since a participating company identified by a unique CoNo value may register more than one set of universal product codes, potentially associated with different

IP-addresses, requiring more than one row in the cross-reference table **215**.

The Company-ID field in the registration template record is used to complete two fields, FromCode and ThruCode, in a row in cross-reference table **215**. These fields specify a range of one or more consecutive universal product codes. Both of these two fields preferably stores a 64-bit integer which specifies a 14 decimal digit universal product code drawn from the global pool of 14-digit numbers which includes the U.P.C., EAN-**13** and SCC-**14** codes. In this way, all three coding systems can be accommodated by the cross-reference table **215**; for example, a 12-digit U.P.C. number 7 12345 12345 9 is the same as the 13 digit EAN number 07 12345 12345 9 and the same as the 14 digit SCC-14 number 0 07 12345 12345 9. If the U.P.C. six digit company identifier 7 12345 is specified in the registration template Company-ID field, the FromCode field of the cross-reference table record would be loaded with the number 71234500000 to specify the lowest valued universal product code cross-referenced to the corresponding IP address in the IPAdr field of the table **215**, and the ThruCode field would be loaded with 71234599999 to specify the highest valued universal product code cross-referenced to that IP address. The use of the low-value/high-value range specification fields in each row of the cross-reference table **215** permits different ranges of universal product codes having the same Company-ID value to be associated with different Internet addresses in the IPAdr field of the cross-reference table **215**, thus enabling a single manufacturer having a single assigned Company-ID value to store information about different products designated by different sets of its universal product codes on different Internet servers, or to cross-reference non-continuous sets of universal product codes to the same or different servers. Note further that a manufacturer need not cross-reference all of its available assigned universal product codes, but may omit unused codes or codes designating products for which no information is to be made available.

The IP-address field in each row of the cross-reference table **215** holds a 32-bit IP address used to route Internet data packets to a destination computer using the TCP/IP protocol. The 32-bit IP address value in the cross-reference table **215** is obtained from the IP address field of the URL table **213**, and that 32-bit address value is prefetched by querying a conventional domain name server (DNS) seen at **217** assigned to the cross-reference resource **101**. The DNS **217** translates the alphanumeric URL in the URL field of the URL table **213** into the current 32-bit IP address used by Internet routers to guide data packets to the proper destination computer. The alphanumeric URL in the URL field of the URL table **213** is supplied via the registration template **207** when Internet location of the manufacturer's product description data is supplied during the registration process.

The separate URL table **213** has a one-to-many relationship to the cross-reference table **215** and uses the 32-bit IP address value as the relational key. This arrangement allows a single URL base address to be shared by a plurality of different manufacturers. Thus, for example, a single Internet service provider (ISP) may act as a shared Internet resource for storing data about a products originating from many different manufacturers. It is accordingly unnecessary for each manufacturer to operate its own server or have its own assigned URL. Instead, a manufacturer may place its product descriptions on any server having an assigned Internet address. Note that it is further unnecessary for the manufacturer to have, or supply, an assigned URL rather than a numerical IP address; however, since corresponding URL's are ordinarily available and easer to remember, and because it may be desirable to later change numerical IP addresses while retaining the same URL, the use of URL's for registration is preferred.

Note also that, because URL/IP address assignments may be added, altered or deleted on a daily basis by the URL assignment authority, updates to the DNS tables should also be reflected by automatic updates to the cross-reference table IP-Address fields. In this way, a change in URL/IP address assignments propagated in the DNS system require no additional action on behalf of the manufacturers to insure the continuing ability of the product code translator to produce the appropriate new IP addresses in response to universal product code queries. If, as discussed later, the Internet domain name system itself is used as the product code to IP address translation mechanism, updating two tables would be unnecessary.

With the foregoing as background, the registration and query/response functions performed by the product code translator may by summarized as follows: each participating manufacturer, or someone acting on its behalf, submits a registration which generates an incoming registration template **207** containing information about the registering manufacturer, including an identification of the universal product codes which designate products for which information is to be made available, together with the URL which specifies the Internet resource which will make that product information available.

The supplied URL is stored in the URL Table **213** and converted into a numerical IP address in the IPAdr field of the URL Table **213** using an available domain name server **217**. This 32 bit IP address is stored in the IPAdr field of the cross-reference record (row) in table **215**, along with a specification of the universal product codes of the products described by information which is available at this IP address, the range of codes being specified by the values stored in the FromCode and ThruCode fields in the new record in cross-reference table **215**.

When an incoming query is received by the query handler **204**, a table lookup function is performed by searching the cross-reference table **215** for a row record or records which specify a set of universal product codes which include the code or codes specified by the query. If matching row(s) are found, the IP-address(es) found in the matching row(s) are returned to the query submitter; otherwise, a special code (such as a zero valued IP Address) is returned to indicate that information for the product code(s) of interest has not been registered.

The Internet resource which acts as the product code translator can additionally perform some or all of the following additional functions:

It can respond to a request for information about a particular participating manufacturer and return to the requester the information in the company table **211** as well as the specification of all of the registered universal product codes assigned to that participating manufacturer and the IP address (or URL) of the location where further information on the products designated by the registered universal product codes may be obtained.

The product code translator can respond to a query containing a designation of one or more universal product codes by identifying the email address of the manufacturer. The product code translator, or any other computer which obtains cross-references between universal product codes and email addresses from the product code translator, may act as an SMTP forwarding agent; for example, forwarding

email which contains a designation of universal product code from a sender to an email address designated by the manufacturer of the product designated by that code. Alternatively, resellers and others may obtain email addresses from the product code translator which can be included in "mailto:" hypertext links in product listings, allowing a webpage viewer to display and complete a blank email request for information which is routed directly to the manufacturer's designated email address. The email address returned in response to a request may be a standard email address such as "upcinfo@domainname" where "domainname" is the domain name portion of the URL supplied by the manufacturer, in which case the specific universal product code would, by convention, be supplied as all or part of the "subject" of the email message sent to that address, enabling the manufacturer to identify the specific product which is the subject of the inquiry.

The product code translator can further provide all or part of the information from company table **211** to provide information about the manufacturer(s) to whom registered universal product code or codes are assigned. Note that, in general, the information which is required or recommended for inclusion with other company information may be limited to that data necessary or desirable to enable the code translator to perform its functions. Other information about the company may simply be placed in an allocated namespace on the manufacturer's server.

The cross-referencing utility can provide the entire contents of its URL table to a requesting computer, such as a search engine which can then perform conventional "web crawler" indexing of the websites specified by the listed URLs and/or IP addresses, thereby generating complete or partial indexes to all or less than all of the products whose product description locations have been registered with the product code translator.

The cross-reference table **215** can be scanned by the product code translator in response to a request for certain universal product codes only; for example, books are assigned EAN numbers which always begin with the prefix number 978 before the company-id value (publisher designation) portion of the International Standard Book Number (ISBN) which makes up the remainder of the EAN number for each book, allowing all IP addresses for information about books to be provided by the cross-referencing server to create a database or index to book information. In the same way, the cross-reference table could be scanned for product codes assigned to a particular manufacturers (e.g. book publishers) to provide a more focused index.

The product code translator, as noted earlier, may facilitate the registration process by providing a website from which HTML registration form pages may be fetched, displayed and completed using a conventional web-browser program. In addition, the product code translator may advantageously make query forms available to permit information request queries to be made directly, as well presenting informational web pages which provide instructions and guidelines for registration procedures, recommendations for the storage of information on registered information resources, and instructions and downloadable software which may be used to simplify and facilitate searches and functions performed at other resources on the Internet which utilize the services provided by the product code translator. When, as discussed later, the Internet domain name service is employed to perform product code or company code translation to an Internet address, the authorized DNS registration authorities can provide informational and registration services to registrants.

Information Publication by Manufacturers

The present invention provides significant advantages and opportunities to manufacturers. Information which manufactures now distribute in other ways can be made immediately available to those who need or desire that information. Examples include text and graphics which describe and promote the sale of each product to potential buyers; product labeling information, some of which may be required to be made available to potential buyers such as product weights and volumes, ingredients, nutritional facts, dosage and use instructions, some or all of which is now included on product packaging and which can be reproduced as mixed text and graphics HTML page for viewing by distributors, retailers, advertisers, catalog publishers, potential customers and purchasers; logos, photographs of products, and other graphics files in a variety of resolutions for use by both electronic and print rendering to promote product sales, usage and support. Instructional and service information including self-help diagnostics and recommended solutions, product part lists and ordering information, product return procedures, current pricing information, identification of dealers and distributors, warranty and guarantee explanations, and support telephone numbers may be provided.

The scope and content of the information each manufacturer makes available is completely under the control of that manufacturer. In order to make this information accessible in a standard way, it is desirable that the manufacturer conform to standard resource naming conventions so that interested parties which obtain the manufacturer's registered IP address from the product code translator can find the desired information at this address. This naming convention may take numerous forms, and the following are merely exemplary:

A root directory named "upcinfo" may be created on each registered computer, and a subdirectory having a name which is the universal product code (expressed as a zero-filled, right-justified fourteen digit number) is created to hold the information concerning the product designated by that universal product code. At the minimum, each such directory includes a product home page named "info.html" which typically provides whatever general product information the manufacturer wishes to place before all interested parties. This product home page may link to additional information related to the product on other pages when appropriate.

By way of example, a product HTML home page for a book would be created by the book's publisher and could include a complete bibliographic citation identifying the title, author, book type (hardcover, paperback, etc.), recommended retail price, ISBN number, number of pages, publication year, etc. In addition, each book's home page might include an imbedded thumbnail image (JPEG or GIF file) of the book jacket, and links could be added enabling the viewer to see additional information concerning that book when available, such as an interview with the book's author, quotes from favorable reviews, book group discussion guides, a table of contents or introductory chapter, etc.

Thus, information uniquely formatted to best advantage by the manufacturer could be made available by accessing a single URL, having the same form for all products, formed by combining the IP-address obtained from the standard by concatenating a prefix and suffix. The suffix has the form:

"/upcinfo/12345678901234/info.html"

where the numerical part of the suffix is the universal product code directory name, and where the suffix is appended to the at the end of the prefix of the form:

"http://23.123.40.198"

consisting of the protocol identifier "http://" and by the 32-bit IP address from the product code translator written in its standard four decimal number format (four three digit numbers separated by periods, each of which is a value in the range 0–255 representing the binary value of one of the four 8-bit bytes making up the 32-bit IP address).

If a manufacturer stores product information in a database, the product directories and the HTML and other data files which are to be made available can be rewritten automatically under program control as the information in the manufacturer's database changes. Alternatively, a request for a particular "file," such as the web page designated "/upcinfo/product-code/info.html," may be intercepted at the manufacturer's server and handled as a database query to which the server responds by dynamically writing an HTML response page using information in the manufacturer's product database. Available database program development tools, such as Microsoft's Access 97 and Borland's Delphi 3.0, include database manipulation tools which allow programs to be readily written which automate the process of generating product description pages from an existing database.

The present invention may be employed to allow the same information found on a product's packaging to be made available to prospective online buyers. For food products, for example, in addition to the product name, logo and promotional materials, such existing packaging information typically includes an ingredient list, nutrition facts, serving suggestions and directions, recommended recipes, and product guarantee information. Over the counter pharmaceuticals, cosmetics and health care products often include further information, such as specific directions on dosage and use, warnings and instructions in the event of misuse, storage and product lifetime information, and active ingredient specifications. Frequently, this valuable information is printed on a product container or container insert which is discarded shortly after purchase. By making this information readily available to purchasers and end-users over the Internet, the manufacturer can help insure that such valuable product information, some of which may be legally required on the product's packaging, is available to the consumer at the time of an online sale and after the product has been purchased.

While information of the kind traditionally placed on product packaging already exists and can be converted by the manufacturer into a format suitable for publication on the World Wide Web, and thus made widely available at little cost, the invention allows information in other forms to be provided at low incremental cost. For example, multimedia presentations may be presented to promote, describe and support a product and its uses. User manuals and service documentation can be provided in Adobe Acrobat portable document format or the like for viewing and printing by resellers, service personnel and consumers.

It is frequently desirable to transfer to another computer data created by the manufacturer which provides limited product description information for each product offered to enable more efficient indexing, cataloging, inventory control, and other applications. By way of example, in the bookselling industry, publishers, distributors, retailers, and libraries often require a database of bibliographic information which consists, for each book, of the book title, author name(s), publisher's name, publication date, type of book (hardcover, paperback, etc.), page count, recommended retail price(s), and ISBN number (which takes the form of a subpart of the EAN universal product code). To the extent the content and format of data records which describe particular classes of products in particular industries and trade groups have been previously adopted and placed in widespread use, those structured data records may advantageously be made available utilizing the present invention. This is preferably achieved in two ways: a data record (file) containing such field-structured information about each product which is designated by a universal product code is placed by the manufacturer in the directory it creates for that product. This structured data record is given a filename indicative of the format used to store the structured data. For example, each directory bearing a name corresponding to the EAN number for a book would preferably contain a file named "biblio.dat" which contains a single structured record containing bibliographic data describing that book.

In addition, the manufacturer would place a combined file, also called "biblio.dat" in its root "\upcinfo" directory which contains all of the records for all of the products individually described in the subdirectories which have that structure in a single file. For most manufacturers, these structured data files, both individual record files in the subdirectories and the combined file in the root directory, may be automatically created and updated on a periodic or dynamic basis from the content of the manufacturer's existing database. The use of a single combined file at each server permits multi-manufacturer databases to be created by first retrieving the IP-addresses of all or part of the cross-reference table **215**, and then retrieving and merging the combined data files from the "/upcinfo" directories from each identified server. Alternatively, when information about all of a given manufacturer's products of a given type is not desired, the needed individual structured data files can be retrieved from the individual product directories.

As described later in more detail, the information which the manufacturer makes available can advantageously be stored using the eXtensible Markup Language (XML), which is also well suited for providing metadata which defines and describes the meaning of the various kinds of information that can be provided about individual products, groups of products, and the manufacturers and distributors from which those products are obtained.

This ability to obtain accurate and up-to-date product information from the manufacturer can substantially reduce the cost to resellers, catalog producers, and database vendors which is traditionally incurred in capturing this data by conventional means. For example, a retailer creating a computerized inventory control system for the first time with previously purchased merchandise may use a conventional hand-held barcode scanner to capture the universal product codes from all goods in inventory, and then retrieve complete and accurate product description records for each product via the Internet using the present invention.

The ability to obtain, update, and verify product description information by accessing manufacturer data can be readily included as callable functions built into inventory control and EDI software used by manufacturers, distributors, and retailers. Institutional "consumers," such as hospitals, government agencies, and libraries, may use the information to build internal databases for internal use.

The structured records noted above are typically, but not necessarily, copied into a separate database which is thereafter manipulated by the requester. Because each copied

database record includes a field containing the universal product code, the ability to obtain and verify data in the remainder of the record from the manufacturer's server is retained. Note that it is possible for the user of the local database to verify, update and add to the product information specified by the universal product code at the time that data is referred to or relied upon. In addition, or in the alternative, the database can be periodically and automatically verified against current data made available by the manufacturer and updated to insure the continued completeness and accuracy of the entire local database.

The present invention enables a computer connected to the Internet to dynamically retrieve arbitrarily large quantities of data about an individual product when needed. This capability makes it unnecessary, and normally undesirable, to copy "content" into a local database which is not needed for structured indexing and retrieval purposes. Thus, again using books as example products, the local database might consist simply of title, author and publisher information to form a searchable local database. This database could be built by first obtaining all of the IP-addresses for universal product codes beginning with "978" (the EAN prefix for books) from the product code translator, retrieving the combined "biblio.dat" file from the "/upcinfo" directory at each IP-address, and extracting the universal product code, title, author and publisher data from these records to form the desired searchable local database. This database may then be rapidly searched to produce an output listing of all books meeting a specified search criteria, and complete information about each of the identified books can then be obtained using the universal product codes.

General product information indexes can be also readily be created by means of conventional "web crawler" indexing engines of the type now widely used to index World Wide Web sites. These indexing engines may scan either the product descriptions created by the manufacturer in the form of HTML or multimedia files, or the structured data files containing fielded information, or both. By limiting the scope of the information indexed to the product information data identified by the product code translator, search results produced by these product indexing systems are less likely to be obscured by references to other, less relevant information which happens to employ the term or terms used in a search request.

The principles of the invention may be applied to particular advantage by online resellers. By making detailed, accurate and up-to-date information about products which are offered readily available to interested prospective buyers, both the reseller and the manufacturer can more effectively promote the offered product to an interested buyer, and the buyer can make a more informed buying decision by obtaining more detailed information which facilitates product comparisons and matching the product's features with the buyer's needs.

In this regard, it may be noted that small retailers can employ shared software and services, and share access to product information and promotional materials made available by the manufacturer in accordance with the invention, at low costs, enabling even the smallest retailer to offer its entire inventory of products (and more) to its customers at low cost, with each product being fully described and promoted by the materials made available by the manufacturer. Similarly, small manufacturers can effectively describe and promote their products throughout a widespread distribution system by simply placing their available promotional and descriptive materials on an available shared server and registering the assigned universal product codes together with the shared server's address, for distribution by the product code translator, all at minimal cost.

In addition, the present invention may be used to advantage in combination with Electronic Data Interchange, a standard mechanism for exchanging business documents in standard format between computers. EDI systems typically use value added networks (VANs), such as the networks provided by GE, IBM Atlantis and Sterling, or EDI transfers can be made via the Internet using services such a those provided by EDI Network of Turnersville, N.J. Using EDI, manufacturers make available electronic catalog descriptions of their products being offered for distribution and resale. When a buyer selects products of interest to order from the vendor's catalog, the retailer's computer accesses the vendor's computer to transfer the U.P.C. codes to the retailer's computer without rekeying. The retailer may then issue an EDI **850** purchase order transaction which is sent to the vendor's mailbox. In addition, the EDI system may transfer limited additional information to the retailer, such as suggested retail price. When the products are shipped, an EDI **856** shipping notice is sent to the retailer containing bill of lading information (bill of lading number, carrier and weight), purchase order information, and carton contents using U.P.C. product codes and counts. The vendor also sends an EDI **810** invoice to the retailer in EDI format which enables the retailer to process the invoice and schedule payment either by check or electronic funds transfer, using an EDI **830** remittance advice transaction to give payment details for invoices being paid.

These EDI transactions enable retailers to not only automate product procurement functions but also to easily maintain an accurate inventory control system in which each product is designated by a universal product code. The present invention may be used to augment an EDI system by providing resellers and consumers with detailed product information for any product designated by a universal product code which is made by a participating manufacturer.

Internet Service Providers, such as the ISP indicated at **111** in FIG. **1**, may provide shared computer services which interoperates with a reseller's inventory control system to provide customers with the information they desire before and after making purchases.

As seen in FIG. **1**, and as previously discussed, a reseller (including both the example distributor **105** and the example retailer **107**) may be assumed to have conventional inventory control systems, typically using EDI document processing, which includes in each case inventory data consisting of at least the universal product code for each product and, typically, count numbers indicating quantity on hand, quantity on order, quantity backordered, etc. This limited part of the reseller's database can be transferred from the reseller's inventory database (at **105** or **107**) to an ISP **111** which serves many resellers but maintains a table of universal product codes for all goods offered by each reseller served, together with the on-hand counts for each code.

The ISP **111** hosts a website for each reseller served in conventional fashion, typically using a domain name assigned to the reseller. The ISP further makes available online merchant software which enables customers to search the reseller's website for products of interest, and view lists of products resulting from each search. Examples of such merchant software include Microsoft Site Server, available from Microsoft Corporation, and Merchantec Softcart marketed by Mercantec, Inc. of Lisle, Ill. Using the present invention, product listings presented to customers by these online merchant software systems may be enhanced with links to detailed information about any product of interest

made available by participating manufacturers. The searchable product database used by the ISP **111** may be built, as described above, using the universal product codes supplied by the retailer to access the structured data files made available by the participating manufacturers (e.g, manufacturer **103** in FIG. **1**) at the IP-addresses supplied by the product code translator.

The implementation of the invention may be facilitated by supporting software which performs a number of utility functions. As noted above, programs may be readily written to automate the conversion of information stored in a manufacturer's existing product database into the form of static or dynamically generated HTML pages which can be transmitted to fulfill information requests routed to the manufacturer by the cross-referencing facility. Industry and inter-industry groups can promulgate standards and guidelines which will promote consistent formats for product descriptions which are accessed in accordance with the invention. Inventory control and online merchant software can be readily enhanced to take advantage of the availability of database records and more robust product descriptions which are made available via the Internet. Product information can be made available at terminals and kiosks placed in retail stores, showrooms and public places.

Using HTTP Relocation to Redirect Product Information Request Messages

The Perl program show.pl, listed in detail in the microfiche appendix, is a CGI (Common Gateway Interface) program which executes on a Web server and which operates as a product code translator as seen at **101** in FIG. **1**. This illustrative program uses a file-based database rather than the relational database depicted in FIG. **2**. The database consists of a set of files, each of which is designated by a file name consisting of a company code followed by the suffix ".xrl" and each containing cross-referencing information for all product codes beginning with that company code. The Perl program show is specially adapted to locate information on books which are generated by a universal product code known as the International Standard Book Number (ISBN), a nine digit decimal number followed by a check character, used by the publishing houses, book distributors, retail bookstores and libraries to uniquely identify books. A variable number of leading digits of each ISBN designate particular publishers, with the remaining digits being assigned by that publisher to designate a particular edition of a particular book.

The Perl program show.pl processes an incoming HTTP message containing a parameter which specifies the value of a universal product code (in this case, an ISBN number), performs a table lookup operation to retrieve the URL at which information about the product specified by that URL may be found, and then returns an "error" message to the requesting browser which contains that URL in the response message's "Location" response field. As specified in Section 10.11 of the Hypertext Transfer Protocol—HTTP/1.0 specification, RFC 1945 (May 1996), the Location response-header field defines the exact location of the resource that was identified by the Request-URL for type 3xx responses, and the location field must indicate the server's preferred URL for automatic redirection to the resource. Only one absolute URL is allowed. The response header may also include the status code **302** which, under the HTTP protocol, indicates that the target data has "moved temporarily" to the URL specified in the location field. In practice, however, it has been found that inclusion of a status code value is not necessary to enable existing web browser programs to automatically redirect the original request to the new location specified in the location response header.

The Perl CGI program processes an incoming HTTP message request directed to a URL of the form "http://www.upclink.com/cgi-bin/show?isbn=1234567890" which is parsed as follows: "www.upclink.com/cgi-bin" is the name of the directory holding the show.pl Perl program and "show?isbn=1234567890" calls the show.pl CGI program and passes to that program the parameter "1234567890" represented by the parameter name "isbn". The ability to execute Perl CGI programs is a common feature of most web servers, and is described, for example, in *"Developing CGI Applications with Perl"* by John Deep and Peter Holfelder, ISBN 0-471-14158-5 (John Wiley & Sons—1996). The Perl programming language is described in many texts including *"Perl 5 Complete"* by Edward S. Peschko and Michele DeWolfe, ISBN 0-07-913698-2 (McGraw Hill 1998). Hypertext Markup Language is also widely used and described, for example, in *"HTML Publishing Bible"* by Alan Simpson, ISBN 0-7645-3009-7 (IDG Books Worldwide 1996).

As seen in FIG. **3**, the show program is entered at **301** and calls a sub routine named get_isbn which processes the incoming message. As indicated at **303**, the subroutine get_isbn first loads the input string variable named $isbn with the parameter named "isbn" supplied by the calling message. The subroutine get_isbn then calls the subroutine isbn_message which returns the string "ok" if the contents of $isbn satisfy the requirements for a correct ISBN (International Standard Book Number); that is, the string must contain 10 digits as indicated by the test at **305** and, as indicated as **307**, the first nine digits must translate using a predetermined algorithm (performed by the subroutine check_char) into a check digit character which matches the last (10^{th}) character in the incoming ISBN number stored as the string $isbn.

The algorithm for generating an ISBN check characters works as follows. First, note that an EAN numbers for books may be converted to the book's ISBN number by removing the first three digits (978) and the last digit from the EAN (the last digit is the EAN check digit, leaving a nine-digit number. For example, EAN 9780940016330 becomes ISBN 094001633 (the first nine digits without the ISBN check character. To generate the ISBN check character, each ISBN digit is multiplied by a predetermined associated weighting factor and the resulting products are added together. The weighting factors for the first nine digits begin with 10 and form the descending series 10, 9, 8 . . . 2. Thus for the nine digits 0 9 4 0 0 1 6 3 3, the products summed are 0+81+32+0+0+5+24+9+6=157. This sum is divided by the number 11. (157/11=14 with 3 remainder). The remainder, if any, is subtracted from 11 to get the check digit. (11−3=8). If the check digit is 10, it is represented by the Roman numeral X. The final ISBN in our example is accordingly 0-940016-33-8. By generating the check digit and comparing it with the received check digit, the validity of the ISBN may be verified.

If the incoming ISBN string passes all of these tests, the routine isbn_message returns "ok," otherwise, it returns an appropriate error message and the subroutine send_error_page is called at **309** to write and transmit an HTML error page to the requester, advising that the ISBN number supplied was incorrect.

When the request containing an invalid ISBN number is being supplied from a source other than the user, such as an online retailer's web site which employs the ISBN number

supplied by its inventory control system, an advisory error message can also be sent directly to the retailer to indicate that an error was detected. Although such an advisory should be unnecessary, since a similar algorithm for identifying invalid ISBN numbers should be used by the retailer to validate the data before transmission, it is nonetheless desirable to report such errors to the source as well. The error report may sent as an accumulated error log file or as an immediately transmitted message sent to a predetermined error message handling routine provided at the retailer's server.

If the $isbn variable meets the length and check-digit tests, it is processed by the subroutine make_cocode which determines, for that ISBN number, how many of the leading digits constitute the "company code" which is assigned to a particular publisher. The routine make_cocode performs this operation by calling the subroutine second_hyphen_ position which performs tests to determine the number of digits in the company code which can be established from the value of $isbn.

The relationship between any given ISBN and the URL which identifies the source of information about the book designated by the ISBN is selected by the party (typically the publisher or its designated agent) which controls the server which provides that information. Because different publishers and their web site hosts may use different methods for establishing URLs for their book information, the Perl script show.pl operates in different ways depending on the company code $cc which forms the leading digits of the incoming ISBN.

The show.pl script handles these differences by fetching a file of control information from a file having the name "../link/cocode.xrl" which is constructed from the company code. The portion "../link/" is a predetermined data directory available to the CGI script show.pl, "cocode" is the numerical string corresponding to the company code extracted at **311** from the ISBN by the make_cocode routine, and ".xrl" is a standard file suffix. For example, if the incoming ISBN is "8870812345" the routine make_cocode will determine that the first five digits are the company code, causing file name "../link/88708.xrl" to be stored as the variable $cstr. As seen at **313** in FIG. **1**, the subroutine load_xrl loads the named control file from local disk space at the cross-referencing server which executes the Perl script. If the file named $cstr is not found, an error report is issued indicating that no cross-referencing data has been supplied by the company identified by the company code in $cstr as seen at **315** and **317**.

The contents of the fetched file are then analyzed by load_xrl which fetches the first line and performs a test at **321** (in FIG. **4**) to determine if the fetched file is a multiline text file with the first line holding the string "list". If not, the file contains a single line.

If the file is a single line file, as seen at **323**, the contents of the single line are used in combination with the ISBN number to form the URL which specifies where the desired book information may be obtained. For example, if the first line of the control file named "../link/88708.xrl" contains the string "hstp://www.upclink.com/ss/.html", the subroutine load_xrl inserts the hyphenated form of the ISBN at the position indicated by the space to form the URL "http://www.upclink.com/ss/88708-1234-5.html". The first and second characters of the control file are used to identify variations in the manner in which the ISBN and the control file string are combined to form the URL. For example, if second character of the control file (an "s" in the example) is an "s", the ISBN is hyphenated before being substituted for the space in the string, and the "s" is replaced with a "t".

If the first line of a multiline control file contains the string "list", all of the lines of the control file are read into a hash table. Each line contains both an ISBN and the URL where information about the book identified by that ISBN can be found. A hash table lookup is then performed to find the particular line holding the target ISBN, and the desired URL is obtained from the line found as indicated at **325** in FIG. **4**.

After the target URL is formed, the subroutine send_ response returns an HTTP response message to the requesting browser which reads:

"Content-type: text/html\n location: $target \n\n" where the URL previously determined is substituted where $target appears. This message is interpreted by the web browser which receives the message as an indication that the requested information (requested from the URL "http://www.upclink.com/cgi-bin/showSisbn=1234567890") has been relocated to the URL specified by $target. The requesting web browser then automatically resends the request to the location where the needed information is actually located, and does so in a way that is transparent to the user who will normally be unaware that the transmitted request has been redirected to a different location.

FIG. **5** of the drawings provides an overview of the typical operation of the CGI Perl script show.pl described above. An online shopper manipulates a web browser application program indicated at **330** to look for and purchase a particular book. In doing so, the shopper provides the browser **330** with the URL of an online book retailer which operates the online sales server seen at **332** in FIG. **5**. During the browsing session, the sales server transmits a web page **334** which lists the citations to one or more books, and each of those citations anchors a hyperlink to the cross-reference server seen at **336**.

The operator of the sales server which creates the book list web page needs nothing other than the ISBN number of the book to create a link. However, as part of its inventory control system database, the retail web site typically has limited additional information, such as the title, author name(s), publisher name, as well as the ISBN. Using this available information, the book listing webpage can include an informative citation to the book which forms the anchor of a hyperlink to additional information. For example, if the following citation information from the retailer's inventory control system is used to display the following book listing:

8 *Ball Chicks: A Year in the Violent World of Girl Gangs,* By Gini Sikes, Anchor Paperback, ISBN: 0-385-447432-6. *LearnMore*

The hyperlink anchor "Learn More" at the end of the citation is formed by the following HTML:

```
<a
href="http://www.upclink.com/cgi-bin/show?isbn=
0553571656"
>
Learn More
</a>
```

When the anchor text "Learn More" on the web page **334** is clicked on by the shopper, an HTTP request message **338** is sent to the href address of the cross-referencing server **336**, triggering the execution of the CGI Perl program show.pl which returns the redirection error message **340**, thereby informing the web-browser **330** that the desired information is at a different location. The web browser **330** immediately (and transparently to the shopper) reissues the request to the address specified in the relocation error message. This redirected request, seen at **342**, is transmitted to the product information server **344** operated by the publisher, which returns a web page **346** to the browser

containing the desired additional information on the book. The web page **346** typically includes an image of the book jacket, a synopsis of the book, and brief reviews, and may well contain links to additional information provided by the publisher, such as an author interview, a table of contents, or whatever else the publisher may wish to include. In this way, the buyer is provided with up-to-date and detailed book information which is equivalent to or superior to the information which may be obtained by picking up the book from the shelf of a "bricks-and-mortar" bookstore.

Image-Cued Links

Because additional information may not be available via the Internet from the publisher for all of the books in the retailer's inventory, it is desirable to provide a mechanism which avoids suggesting that a site visitor should click on links that will ultimately prove unworkable.

The mechanism involves the prior retrieval of data describing which universal product codes have been registered before web pages containing links using these codes are generated. For example, a sales website could transmit a list of the company codes for those universal product codes of products to be offered for sale, and obtain in return a listing of those company codes which have been registered. Alternatively, a list of supported company codes could be periodically broadcast (e.g. by FTP transmission) to subscribers. The sales website could then use this list to distinguish those products for which additional information was being made available by the manufacturer from those for which there is no additional information, and include links on product lists only when they will work.

Alternatively, the web page producing site can perform a prior fetch of the needed Internet addresses from which product information may be obtained by sending a request message containing one or more universal product codes (or company codes) to the cross-referencing server and receive in return a list of the corresponding Internet addresses. In this way, the links to additional information contained on product list pages can be refer to the manufacturer's servers directly, and can suppress the creation of links when no information has been made available.

A mechanism here called "Image-cued links" can also be used to suppress the appearance on a web page of links to unavailable product or company information. For example, an image-cued link can display a graphical icon for a book listing on web page **334** which might be either a visible button with the legend "Learn More" or an invisible (transparent or single-pixel) graphic image, depending on whether or not the ISBN for a listed book has a corresponding URL stored at the cross-referencing server **336**. The HTML for an image-cued link to information about the book identified by ISBN 0821219804 might be written like this:

```
<a
href="http://www.upclink.com/cgi-bin/show?isbn=
0821219804">
<img
src="http://www.upclink.com/cgi-bin/button?isbn=
0821219804"
border="0" width="51" height="21"
>
```

where the "anchor" for the link to the CGI script named "show" is an imbedded image (fetched from a different CGI script named "button" which also executes on the cross-reference server). The Perl script button.pl is also reproduced in the Appendix and performs the same initial processing of the incoming ISBN number as the script show.pl. Both show.pl and button.pl determine whether data is available from which a cross-reference from an incoming ISBN number to a URL can be made. If it the cross-reference can be made, show.pl returns a relocation message containing the needed URL whereas button.pl returns the URL of an image that indicates that more information is available. If the cross-reference cannot be made, show.pl returns an error message in the form of an HTML page while button.pl returns the URL of a null image (either a transparent image or a single pixel image). Note that the content of the image file may be controlled by the web page producer since the CGI routine at the cross-referencing server (e.g. button.pl) may be unique to the caller and may hence return image URL's specified by the caller. Alternatively, a single CGI button routine can be used with the desired image URLs being passed as parameters to the cross-referencing server.

The net effect on the web page is the appearance of a button or other image inviting the site visitor to click on a link to learn more when more information is available, but to suppress the display of the button, or to display a "no information available" button or image when the cross-reference cannot be made. In this way, the site visitor is affirmatively informed when more information is available, and discouraged from looking further when no information is available, while the HTML placed on the book listing has a standard form which can be included without prior knowledge of whether needed data is available or not.

Product Code Cross-Referencing Using the Lightweight Directory Access Protocol

The Internet LDAP protocol may be used to advantage to implement the product code translation process. This protocol, developed at the University of Michigan and later further developed by Netscape Communications Corp. provides both access and update capabilities, allowing directory information to be created and managed as well as queried. LDAP is an open Internet standard, produced by the Internet Engineering Task Force (IETF), the same body responsible for creating TCP/IP, the Internet domain name service (DNS), and the hypertext transport protocol (HTTP). The LDAP protocol is defined in RFCs 1777 and 1778 and informational documentation is further provided in RFC 1823. The use of LDAP to provide directory lookup services via the Internet is further detailed in the literature. See, for example, *Implementing LDAP* by Mark Wilcox (Wrox Press—1999) and *LDAP— Programming Directory Enabled Applications with Lightweight Directory Access Protocol* by Tim Howes and Mark Smith (Macmillan Technology Series—1997). Operational LDAP server software may be purchased from a variety of sources, and includes the "Netscape Directory Server" marketed by the Netscape Communications Corporation.

An LDAP server may be advantageously employed to store "entries," each of which is uniquely identified by a distinguished name (DN) which may take the form of the company code portion of the universal product code, creating a "flat namespace" in a single level tree structure, with the remainder of the entry including a string specifying the URL of the server resource from which information about products assigned that company code may be found. In one arrangement, an online merchant's server may send a request to a remote directory server using the LDAP protocol to obtain the URL at which information about a specific product is available. Next, the merchant's server could again use the LDAP protocol to fetch information about a specific product designated by the remainder of the universal product code from a second LDAP directory server at the URL specified by the first server, the second LDAP server being

operated by the product manufacturer to store the URL at which data describing particular products is stored. The actual product data may advantageously be stored as XML "documents" as discussed later.

Product Code Cross-Referencing with Domain Name Servers

Cross-referencing a universal product code to the Internet address of the source of information about the product designated by that code can be advantageously performed by Internet domain name servers (DNS). Conventional Internet domain names are symbolic names (a character string) that identify different computers and resources on the Internet. While computers connected to the Internet actually use binary IP (Internet Protocol) addresses to find each other, people find words and abbreviations to be much more convenient to remember and use. The Internet domain name system assigns computers and resources a domain name that corresponds to the numerical IP address used to access that computer or resource. Each domain name must be unique, and server operators register their desired domain names with a domain name registration service.

Domain names are composed of a hierarchy of names that appear in descending levels from right to left. Therefore, the levels that appear at the end of URLs and E-mail addresses are the first-level and second-level domains. For example, in the domain name "patentsoft.com," the suffix "com" is the first-level domain, and "patentsoft" is the second-level domain. By creating a new first-level domain (e.g. "upc"), the universal product code, or the company code portion of a set of universal product codes, could form the second-level domain.

Note that, if the company code portion universal product code is used as the second-level domain, registration need only be done once for all product codes sharing that company code. Note also that, as noted earlier, it is desirable that each manufacturer respond to a request for information about that particular participating manufacturer. For example, retail merchants and distributors may advantageously use the company code portion of a universal product code to access a variety of useful information about the company generally, including contact information and distribution, shipping and discount policies. In this way, any retailer can use the web to obtain general information about a company while those retailers with established accounts with a particular vendor (as confirmed, for example, using digital signatures) may obtain private information which is hidden from the general public.

In this way, universal product codes and/or company codes can be used as domain names which are cross-referenced to IP addresses using existing DNS facilities. Thus, when a web browser issues a request directed to a URL including the domain name "123456.upc," the DNS server (typically assigned by the customer's Internet service provider) responds with a corresponding IP address of an information server maintained by or for the company designated by the company code "123456" (assuming that company has registered the IP address for that company code with a DNS registration authority). If the assigned DNS server doesn't already have the cross-reference between that product code or company code domain name and the manufacturer's server's IP address, it asks the primary DNS server that is responsible for the domain if it has the server's IP address. If the primary DNS is busy or unavailable, it will ask the secondary DNS server assigned to that domain. When the customer's DNS server gets the manufacturer's server's IP address, it can then ask that server for permission to view a web page or otherwise obtain information about the designated product.

Note that the registered domain name can advantageously take the form of the company code only, with the remainder of the universal product code being passed as a parameter to the manufacturer's server. For example, if "123456" is the company code portion of the product code, and the trailing digits "7890123" designate a particular product made by that company, the assigned domain name might be "123456.upc" and the full URL for the product might take the form "123456.ean/7890123" (for a thirteen digit EAN-13 code). It would thus be up to the manufacturer's server to intercept and process the product designating suffix digits "7890123" to identify and return information on the particular product specified. Note also that different universal product code systems, such as the UPC codes used in the United States and Canada, the EAN codes used elsewhere in the world, a 14 decimal digit universal product code drawn from the global pool of UPC and EAN numbers, or ISBN numbers used by the publishing industry, might conveniently be assigned different first level domain designations, such as "upc," "ean," "gpc," and "ibn" respectively. At the time the IP addresses corresponding to a given company code and/or product codes are registered, appropriate procedures may be used to confirm that the applicant for DNS registration has been assigned that particular code in the existing uniform product code system, or has authority to act on behalf of the true assignee of that code. In this way, the existing universal product code registration authority retains primary responsibility for assigning codes, whereas the DNS registrar at most need only confirm the identity of the DNS registrant.

The Internet domain name system that is currently in widespread use is described in RFCs 1034 and 1035. RFC 1034 provides an introduction to the Domain Name System (DNS), and omits many details which can be found in its companion RFC 1035 which is entitled "Domain Names—Implementation and Specification." Recently, The Internet Corporation for Assigned Names and Numbers (ICANN), a non-profit corporation, was formed to take over responsibility for the IP address space allocation, protocol parameter assignment, domain name system management, and root server system management functions currently performed under U.S. Government contract by LANA and other entities. Accordingly, at this writing, ICANN, or a new authority authorized by ICANN, would be the appropriate authority to reserve DNS namespace for the translation of universal product codes (or company codes) to ip addresses as proposed here.

The use of the domain name server system as the mechanism for cross-referencing universal product codes and Internet addresses is used in the illustrative embodiment of the invention which is depicted in FIG. **6** and described next.

Using DNS, XML, XSL/CSS and XPointers

FIG. **6** of the drawings illustrates a system which permits computers operated by large numbers of manufacturers and large numbers of online retailers to work together to provide shopping services to customers via the World Wide Web.

FIG. **6** shows a browser **410** being used by an online shopper to view products offered by a retailer which operates an inventory control system computer seen at **422**. The inventory control system **422** is conventional and includes one or more conventional point of sale registers as illustrated at **421** through which sales are made to customers who visit the physical "bricks and mortar" store. In addition, however,

the inventory control system is provided with a "web register" which, to the inventory control system appears to function in the same way as a conventional point of sale register but which, in fact, operates through a sales server **430** which provides Internet services on a shared basis to multiple retail stores and their inventory control systems. A communications pathway connects the web register **420** and the inventory control system **422** to the shared sales server **430**, with the inventory control system **420** supplying the product codes and corresponding on-hand quantities as indicated at **432** and receiving from the shared sales server order information as indicated at **434**.

A product information server **423** supplies product information to the browser **410** in the form of XML data as indicated at **424** in response to requests **425**. The browser **410** preferably utilizes the Internet domain name system as proposed above to convert incoming universal product codes into Internet addresses, with the domain name system consisting of an assigned domain name server **470** which receives universal product codes in address requests **472** and returns the registered Internet addresses **474** to the browser **410**. When needed, the assigned domain name server **470** obtains the registered cross-references between universal product codes and IP addresses from the primary DNS **476** or from the secondary DNS **478**.

The shared sales server **430** sends web pages **438** containing information about products available from a connected retailer to the browser **410**, along with XSL (Extensible Stylesheet Language) or CSS (Cascaded Style Sheets) style specifications as seen at **440**. The use of XML and XSL or CSS provides several advantages. First, the selection and rendering of the product information is controlled by the links specified in the web page **438** as produced by the sales server. For example, if the web page **438** contains a product listing web page created in response to a search request from the browser **410**, each included product description may include a link to only an that portion of an XML product description which contains a brief product description and a thumbnail image of the listed product, whereas, in response to a customer's request for more detailed information, the sales server may return a web page containing an XML "Xpointer" link to detailed product information and/or to an enlarged image of the product. In both cases, the style in which the XML data is rendered by the browser (e.g. typeface, font size and color, background color, etc.) is controlled by the style specification supplied by the sales server **430**. In this way, the same XML data may have different visual styles when included on the pages created by different retail vendors.

The XSL (Extensible Stylesheet Language) consists of consists of two parts: (1) a language for transforming XML documents, and an XML vocabulary for specifying formatting semantics. An XSL stylesheet specifies the presentation of a class of XML documents by describing how an instance of the class is transformed into an XML document that uses the formatting vocabulary. XSL is described in the World Wide Web Consortium's "Extensible Stylesheet Language Specification," the current working draft of which may be found at http://www.w3.org/TR/WD-xsl (Apr. 21, 1999). An XSL stylesheet processor accepts a document or data in XML and an XSL stylesheet and produces the presentation of that XML source content as intended by the stylesheet. It is contemplated that most major web browser applications will include XSL stylesheet processors which will enable them to convert the combination of XML and XSL data into a form, such as a viewable HTML web page, as specified by the XSL.

In addition to using XSL to specify the rendering style of XML data, cascaded style sheets (CSS) can also be used as set forth in the Proposed Recommendation dated Apr. 28, 1999 from the World Wide Web Consortium (see http://www.w3.org/TR/1999/xml-stylesheet-19990428). This specification allows a stylesheet to be associated with an XML document by including one or more processing instructions with a target of "xml-stylesheet" in the document's prolog. The World Wide Web Consortium's recommendation regarding cascaded style sheets may be found at http://www.w3.org/TR/REC-CSS1 (Jan. 11, 1999) which specifies level 1 of the Cascading Style Sheet mechanism (CSS1). CSS1 is a simple style sheet mechanism that allows authors and readers to attach style (e.g. fonts, colors and spacing) to HTML documents. The CSS1 language is human readable and writeable, and expresses style in common desktop publishing terminology. One of the fundamental features of CSS is that style sheets cascade; authors can attach a preferred style sheet, while the reader may have a personal style sheet to adjust for human or technological handicaps. Thus product descriptions as viewed on the browser may include content from the product manufacturer, reflect a preferred rendering style specification from the online reseller, as well as the personal style preferences of the viewer.

XSL could alternatively be used at the shared sales server **430** to transform XML data fetched by the server **430** from the manufacturer's server **423** and then converted into HTML documents with CSS style sheets at the sales server **430**. This has the benefit of being backwards compatible with browsers which do not include the ability to handle XSL/CSS. Alternatively, XSL conversion can be performed on the server **430** to transform XML data into XML documents with CSS style sheets. XML, unlike HTML, comes with no formatting conventions and will always need a style sheet to be displayed. This method requires that the browser have the ability to use CSS to render XML. A third alternative is to use XSL to generate HTML/CSS on the client side, a method which requires that the browser have the ability to directly use CSS and XML, which older browsers cannot do. Finally, the browser may transform XML and XSL into "CSS formatting objects". Compared to the previous method, this method is more direct as the content isn't converted to/from HTML.

The ability to select only a portion of an XML product description document for reproduction on a web page is provided by the Xpointer protocol. As explained in the World Wide Web Consortium Working Draft of Mar. 3, 1998, at http://www.w3.org/TR/WD-xptr, the XML Pointer Language (Xpointer) document specifies a language that supports addressing into the internal structures of XML documents. In particular, it provides for specific reference to elements, character strings, and other parts of XML documents, whether or not they bear an explicit ID attribute. Using Xpointer, only selected portions of an XML product description made available from the manufacturer's server need be presented on a given web page, enabling the creator of the web page which links in XML data to control the nature and extent of the information shown.

The manner in which explicit relationships between two or more data objects, such as a retailer's product list page and the product information about a product listed on that page, may be expressed as a link asserted in elements contained in XML documents. These "XLinks" in the simplest case are like the HTML links described above in that they are expressed at one end of the link only, are initiated by users to initiate travel to the other end of the link, go only

to one destination (which may be determined by a DNS server or by an independent cross-referencing server), and produce an effect which is mainly determined by the browser. The functionality of links is being vastly extended, however, by the XML Linking Language (XLink) specification being developed by the World Wide Web Consortium and available at http://www.w3.org/TR/WD-xlink. As extended, the XLink specification will provide more sophisticated multi-ended and typed links which can be used to advantage to automatically incorporate linked-in product information from one or more manufacturers into displays and multimedia presentations presented by retailers and others.

As previously discussed, in addition to the use of a product code translation utility which cross-references all or part of a universal product code into an Internet address, it is desirable to establish a protocol or convention which enables a requester to specific kinds of information about identified products or companies. In a conventional HTML system, this can be done by establishing naming conventions, as described above, for selectively locating different kinds of information about a product designated by a given universal product code, as well as different kinds of information about the company identified by the company code portion of the product code.

The metadata capabilities of XML can be used to advantage to provide an extensible system for dividing product and company information into a hierarchy of nested named elements which can be selectively accessed. Using the Document Type Descriptor (DTD) component of XML, the makeup of the required and optional components of such information can be defined in a standard way, facilitating the definition and validation of data structures to be used on various classes of products.

The World Wide Web Consortium has further defined the "Resource Description Framework (RDF) and Syntax Specification" as described at http://www.w3.org/TR/REC-rdf-syntax. RDF provides a foundation for processing metadata (i.e. "data about data") and provides interoperability between computers that exchange information on the Web. Using RDF, data about products and companies, which can be accessed in accordance with the invention by using universal product codes; can be used by search engines to provide access to such information, can be used to automatically catalog the content and content relationships at particular web sites, pages or libraries; can be used by intelligent software agents to facilitate the sharing and exchange of information about companies and products. Using RDF with digital signatures, the privacy preferences and policies of the owners of product and company information can be selectively protected to help build the "Web of Trust" needed for electronic commerce.

Importantly, RDF provides a mechanism for defining metadata in a class system much like the class systems used by object oriented programming and modeling systems. Classes are organized in a hierarchy, with a collection of classes used for a particular purpose (such as the collection of classes describing a "product" and/or the collection of classes describing a "company") being called a "schema." RDF thus offers extensibility through subclass definition. For example, creating subclasses for a particular kinds of product (e.g., publications, software, foods, clothing, etc.) requires only incremental modification of a base "product" schema, and each such subclass may then be further modified to form descendant schema for even more particular kinds of product (e.g., magazines, video games, cereals, shirts, etc.). The shareability and extensibility of RDF also allows metadata authors to use multiple inheritance to mix definitions, providing multiple views of their data, and leveraging the work done by others. From a practical standpoint, the creation of a simple and generic product and company description base schemas which can thereafter be extending using RDF allows basic information about products and companies to be made available early, allowing more elaborate schemas to evolve as experience with the simpler system suggests their utility.

Using these techniques, the product manufacturer may be largely freed from concerns about web page design, formatting and integration with other information, and may concentrate on providing accurate and up-to-date text descriptions of its products, along with whatever images best describe the product, simply by registering the relationship between the manufacturers company code and/or universal product code(s) with the appropriate authority, and following the established content specifications for the information which the manufacturer makes available at the registered IP address.

It should be noted that XML, XSL, Xpointer, XLink and related Internet protocols and specifications are still being defined and extended, a process that can be expected to enhance the ability of the present invention to selectively and attractively provide information about products to those who desire that information from the most knowledgeable and reliable sources of that information.

The Shared Sales Server and the "Web Register"

Conventional retail stores display merchandise for purchase in physical showrooms for inspection and purchase by consumers. These stores typically obtain the products they sell from a large number of manufacturers, either directly or through distributors. Computerized inventory control systems are used to keep track of orders and sales in an effort to insure that goods are available for prompt delivery to customers when purchased while minimizing the expense of the maintaining an unsold inventory of products. Barcode checkout and EDI (Electronic Data Interchange) systems are commonly used in combination with inventory control systems to automate sales transactions. Customer checkout counters equipped with barcode readers automate the customer sales transaction, reducing errors and reducing costs. EDI, the computer-to-computer exchange of business documents in a standard format, automates transactions between the retailer and its suppliers by transferring sales documentation in electronic form, including the purchase order, the invoice, and the advance ship notice. Barcode readers and EDI services are now commonly used even by smaller retailers and have significantly reduced the cost of doing business.

In recent years, the Internet has produced new on-line sales mechanisms which promise to revolutionize retail sales. Using the World Wide Web, both manufacturers and resellers offer products to consumers for direct purchase with product delivery normally being accomplished by mail or a commercial delivery service. The consumer typically employs a web browser to search for products of interest and display product descriptions. Customers make purchases by completing one or more displayed forms to provide needed information (e.g. a credit card number and a shipping destination address).

Many successful web resellers take advantage of the fact that a web presence is inherently world-wide, and that the delivery charges of many services are largely independent of distance. Because a physical showroom is an unnecessary

expense, web resellers can often successfully sell at much lower cost directly from one or more warehouses. To be successful, however, such a web reseller must make its presence known to a large audience. As a result, large and well-funded web resellers capable of effectively marketing on a nationwide scale have done well while those which have attempted to market their offerings locally or not at all have predictably been much less successful.

Large chain resellers with many retail outlets have also established successful web sales operations. Because the cost of creating and maintaining a web presence, as well as the cost of nationwide marketing, can effectively be shared among many retail outlets, and because much of the infrastructure needed to provide order fulfillment is already in place, such chain stores compete well with the nationwide web-only merchants.

Although customers would often benefit from the convenience of reviewing and ordering products from a local reseller, individual retail stores and smaller, local chains have typically been unable to justify the cost of creating and maintaining a web sales system. In addition to the substantial cost of installing and maintaining the needed hardware and software, the website owner must expend significant effort to create and periodically update the product descriptions which customers require in order to make informed purchases. The annual cost of operating a merchant website often far exceeds the revenue which the local merchant would derive from the limited probable volume of online sales.

The infrastructure depicted in FIG. **6** permits smaller and mid-sized merchants to profitably and efficiently offer online ordering services to their customers. By using the cross-referencing capability provided by the present invention, which provides customers with reliable product information direct from the manufacturer, and the shared sales server **430** and web register **420** which provides shareable functions which effectively connect the retailer's existing inventory control system **422** to the World Wide Web, the cost and complexity of retail web sales is largely eliminated, and the consumer obtains the convenience of shopping by computer with the other advantages of shopping with a local merchant.

The shared sales server **430** operates in the same way and provides the same functions as a conventional online merchant server: it enables customers to perform searches of available products and produces listings which the customer can review, with the ability to click on individual items to activate links to additional product information from the manufacturer's server as noted earlier. In addition, the shared server provides "shopping basket" and credit card transaction services to enable the customer to complete purchases.

The shared server **430** and the web register module **420** added to the retailer's existing inventory control system **322** maintain a connection via the Internet or a dial-up modem pathway which permits the inventory control system **422** to upload to the shared server **430** changes to the products (specified by universal product code) being offered for sale, and the quantity on hand. Each time any sale is made by any point of sale register **421** in the physical retail store or by the web register **422**, the quantity on hand value associated with the sold product's code is altered. Similarly, when stock is replenished, the inventory control system **422** reflects the increased quantity on hand. The quantity on hand information passed as message information at **422** permits the shared sales server to maintain a database for each retailer served which indicates the products available for sale and the quantity on hand. When the quantity on hand equals or exceeds the quantity ordered, the on line order is accepted and passed at **434** from the shared server to the web register module **420** which posts the sale in the same way that a point of sale register posts a sale. The fact that the web register **420** performs the same functions as a conventional cash register enables the conventional inventory control system **422** to function in the normal way, with the exception that it must also update the product code and quantity on hand data maintained by the shared server. The fact that the shared server thus "knows" the inventory status allows the shared server to accurately inform the customer when shipment can be expected for goods on hand and when goods which must be replenished will be shipped with a delay. Orders sent to the inventory control system at **434** include the specification of products sold (by their universal product code designation) and the quantities of each sold, as well as address information for billing and shipping. Credit card transactions are handled on a shared basis using standard ecommerce software, either by sending encrypted credit card and other billing information to the retailer for handling, or actually performing the monetary transaction with the customer in its entirety on the shared server, and sending periodic payments and accounting records to the retailer.

Importantly, functions better performed by the retail store using resources best shared with the physical store's operations are not performed by the shared server **430** but rather by the store's inventory control system **422**. Such functions include order fulfillment, inventory pricing, vendor ordering, reordering and payment, and warehouse management functions.

Examples of merchant server software which may be used to implement the shopping basket and credit card transactions performed by the shared server **403** include the SoftCart system offered by Mercantec, Inc. SoftCart 4.0 includes integration to the CyberCash system for credit card transactions and includes a Software Developer's Kit to enable a programmer to integrating SoftCart with WWW programs and legacy systems. A discussion of available systems and techniques is found in *Designing Systems for Internet Commerce* By G. Winfield Treese and Lawrence C. Stewart ISBN: 0201571676 (Addison Wesley 1998).

In accordance with the invention, neither the shared server **430** nor the retail inventory control system **420** need store or maintain descriptive information about products. When an online customer desires information about a given product, it is obtained for review by the customer using XML Xpointer links to the manufacturer's server. XML information is located by supplying the universal product code for the product to the DNS server **472** which stores product code to Internet address conversion information. Product searches which locate products offered by a given retailer (as revealed by the product code and quantity on hand database) are displayed to the customer and, to the extent they result in purchases, are manifested by the transmission of a completed order to the retailer's inventory control system.

In a typical shopping transaction performed on the system shown in FIG. **6**, a shopper employs the browser **410** to visit a web site which executes on the shared sales server **430**. When the shopper reviews an HTML web page which lists available products, as illustrated at **438** in FIG. **6**, the shopper can request additional information on any listed product by using the browser **410** to click on a link anchor on the page **438**, thereby issuing are request **472** to the domain name server **470** for the IP address at which additional information on that product identified by a universal

product code which forms part of the URL contained within the request **472**. The DNS **470**, consulting a primary DNS **476** or secondary DNS **478** if necessary, returns the IP address to the browser **410** which then issues a request **480** for XML product information to the product information server **422** maintained by the manufacturer of that product. Using the XSL stylesheet specification **440** supplied by the shared server **430** (in accordance with the background colors, font styles, etc. which may be characteristic for the web presence of the specific retailer), the browser presents the product description contained or specified by the XML data **424** to the user.

If the user decides to purchase the described product, the "shopping basket" functions of the shared sales server **430** are used to complete the order. Because the shared server **430** maintains a database for that retailer containing the quantity on hand values for each product offered by that server, the customer can be immediately informed if the shipment cannot be made whereas, if the product is available at the retailer's store or warehouse, the online customer's order can be confirmed for prompt delivery. When the order is completed by the shared server **430**, the order **434** which includes the identification of the customer (name, shipping address, etc.) and the identification of the products sold (universal product codes plus quantities sold) is transmitted to the retailer's inventory control system **420**. As explained in more detail below in connection with FIG. **7**, the shared server **430** adjusts the quantity on hand values in its database, and the inventory control system **420** updates its database, with a cross check between the two being made if desired to insure consistency and synchronization.

It is important to observe that the arrangement shown in FIG. **6** isolates the retailer from substantially all of the concerns normally associated with the creation and maintenance of an online sales website. In accordance with the invention, the retailer need not be concerned with the creation or maintenance of accurate information on the products sold, since that task is appropriately born by the product manufacturer which has that information. Similarly, the retailer need not be concerned with the creation, hosting, and maintenance of a reliable and easy to use online shopping experience, a task performed by the shared server **430** for many different retailers. To the retailer, the shared server presents an interface to the inventory control system **420** which behaves much like conventional point-of-sale terminals (cash registers) indicated at **421**.

This "web register" capability can advantageously be installed and maintained, with training being provided to the retailer, by inventory control system vendors as part of the normal inventory control system software, maintenance and training. The shared server functions can advantageously be made available by an independent Internet service provider (ISP) using standard e-commerce software, and the independent vendor of the shared sales server software can provide interface specifications and software support to the inventory control system vendors, enabling them to add the necessary "web register" interface functions to new or existing inventory control systems. In this way, each function is performed by an entity which is already experienced in that phase of the overall system requirements, and each function is provided at little additional cost over costs already born by each contributor.

The relationship between the shared sales server seen at **403** in FIG. **6** and the retailer's inventory control system seen at **420** in FIG. **6** is shown in more detail in FIG. **7** of the drawings. As discussed above, the remote shared sales server operates, from the standpoint of the inventory control system, much in the same way as a conventional point of sale terminal. The other components of the inventory control system include essentially conventional purchasing, receiving, order processing, shipping and invoicing functions described in the literature. See, for example, the texts *Best Practice in Inventory Management,* by Tony Wild, John Wiley & Sons; ISBN: 0471253413 (March 1998) and *Inventory Control and Management* by C. D. J. Waters, John Wiley & Sons; ISBN: 0471930814 (June 1992).

Within the inventory control system, the purchasing module **501** presents printed reports and screen displays which assist purchasing agents to see which products need to be ordered and which vendors need to be contacted to follow up on prior orders, and automates back orders and reorders. The module **502** generates purchase orders, and alerts purchasing agents of urgent or routine product ordering needs by evaluating supplies on hand and estimating the demands for each product to determine if supply levels will fall below the predetermined minimum stock quantities established for each product by the merchant. The purchasing module **501** further creates planned orders based upon a minimum order quantity, order multiple, yield percentage and maximum plan quantities established by the merchant, and alters the purchaser of low product levels automatically. Purchase Orders are generated using screens presented to the purchasing agent which can recall vendor price lists, including price breaks and special promotion pricing. These purchasing functions are typically performed in a bricks-and-mortar retail store or group of stores, and need not be altered to support online sales as contemplated by the present invention.

Similarly, the receiving module **502** accepts inventory control information by capturing data describing incoming orders from vendors, issues inbound receipts and purchase order confirmations, and tracks back orders. Inbound receipts are generated from existing Purchase Orders

The order fulfillment unit **503** accepts both conventional sales orders of the type received by mail or telephone, and processes online orders from the shared server (seen at **430** in FIG. **6**) in the same way. The unit **503** maintains the visibility of those orders until shipment is completed by identifying orders that need attention, including orders that must be rescheduled or expedited, to insure prompt delivery and to provide order status information to the customer. Customer contact information (phone number, email address or mailing address) from the online shared sales server **430**, like similar information obtained by telephone or mail, may be used to automatically issue email notifications or assist the merchant in contacting the customer in other ways to confirm or reschedule orders.

The shipping module **504** is also conventional, and handles outbound shipments, accepts new customer shipping and billing address information, handles partial shipments by identifying items reserved for later shipment, and prints packing slips and bills of lading.

The invoicing module **507** provides invoicing, billing and charging capabilities, printing invoices to be sent to customers. The shared sales server may support billing in several ways: it may simply send orders to the invoicing module **507** including shipping information supplied by the customer using HTML forms; it may verify and accept credit card information and transmit that to the invoicing module **507** so that the actual credit card transaction is between the customer and the retailer, or it may complete the credit card transaction at the shared server, forwarding collected funds together with accounting information to the retailer on a periodic basis.

It is a principle feature of this aspect of the invention that the retail merchant, who already maintains a physical inventory and/or a distribution relationship with manufacturers, as well as an inventory control system for managing its inventory and distribution functions, performs the order fulfillment function using facilities which are shared with those used by conventional "showroom" sales facility. The shared sales server merely processes data, and need not be concerned with the actual selection, purchase or distribution of physical products, nor with the creation of the detailed product information needed by the consumer when making online purchases. Both of these functions remain where they are best performed, with the retailer and the manufacturer respectively.

When sales are performed using conventional point of sale terminals as indicated at **508**, a sales module **510** processes the information. The point of sale terminals **508** perform automated sales checkout using a bar code reader to reduce errors and speed customer checkout times, enabling salespeople to focus more on customer service. Sales are manifested by the identification of goods and quantities purchased, and these are reflected in the actual delivery of goods at the point of sale terminal which, in turn, are posted to decrement the on-hand quantity values for the products maintained in the inventory control database **500**. If selected products are unavailable at the point of sale, a terminal **508** may commonly be used to place an order for future delivery to an identified customer by posting the order into the database **500** for handling by the purchasing module **501**, receiving module **502**, order fulfillment module **503**, shipping module **504** and invoicing module **507**.

Incoming orders from the shared sales server placed on a "web register" are processed in substantially the same way. The order information is obtained from the customer, typically using HTML forms, by the web sales unit seen at **521** in FIG. **7**, and the resulting order in the form of customer information, product identification and quantity sold, and related order information is transmitted via the communications interfaces **531** and **532** at the sales server and the remote inventory control system respectively. The communication link **544** advantageously takes the form of an Internet link between the retailer and the remotely located sales server which operates on a transaction by transaction basis. Note that, because both product identification data and on-hand quantity information is available in the database **540** at the shared server, the two can continue to work without continuous connection, with orders taken at the "web register" being transmitted on a batch basis, and with changes to the database of offered product universal product codes and on-hand quantities being posted on a batch basis from the inventory control system to the shared server.

The sales module **510** processes orders from both the point of sale terminals **508** and from the shared sales server. It packages the customer information and the product and quantity sold information for handling by the remaining modules as discussed above, and schedules multiple deliveries as needed when an order can only be partially fulfilled. When web sales are made, on hand quantities are immediately reserved and, if desired, sales of the same products from inventory via the point of sales terminals **508** can be inhibited (although, in practice, if removal of items from the showroom is needed to fill web orders, that should be done promptly to avoid customer confusion).

The relationship between retail stores, manufacturers and distributors, product information services and consumers, and the computers connected to the Internet which utilize the invention to serve each of these entities, is depicted in FIG. **8** of the drawings.

Consumers and the general public access information and perform transactions via the Internet using conventional web browsers (i.e. conventional web browser application programs executing on desktop computers or workstations) as exemplified at **803, 804** and **805**. Such web browsers typically employ a shared Internet Service Provider (ISP) as indicated at **807** which provides a connection to the Internet **810**.

These consumers may view product information and perform sales transactions as contemplated by the present invention by viewing information made available by retail stores **811, 812, 813** and **814** via a shared sales server **820**, as well as by a server **825** operated by a single retail store (or chain of stores). The servers **820** and **825** transmit web pages to the web browsers **803–805** which include links to product information which is made available by manufacturers **831, 832** and **833** via a shared product information server **840**, and from a server **845** operated by a single manufacturer.

The manufacturers preferably provide product information to their connected server in the form of well-formed eXtensible Markup Language (XML) documents which may be validated against a standard Document Type Definition (DTD) to which all such product information documents should conform. The schema to which such documents adhere may be advantageously expressed in the Resource Description Framework (RDF) and Syntax Specification, as noted earlier, to facilitate the evolution of standardized content definitions for product and company information. The shared product information server illustrated at **840**, in its simplest form, does nothing more than make Internet accessible data storage space available where smaller manufacturers without their own servers can make product and company information available via the Internet. Even the smallest manufacturer can thus make product and company information available to consumers and retailers worldwide at an insignificant incremental cost. Many such small manufacturers can simply use web hosting space provided free of charge by their existing Internet Service Provider for product and company information storage.

Manufacturers are not, of course, restricted to providing such product information through links from web sites of retailers and others. A manufacturer may use the same information to support its own promotional web site offering such things as product directories, press releases, direct sales to consumers, and any other function and service typically provided by a manufacturer's web site. Indeed, by using a predetermined URL syntax, such as http://cocode.ean/homepage (where "cocode" is the company code assigned to that manufacturer), the home page of the manufacturer of any product can be readily accessed if that product's universal product code is known.

The availability of company information, which may be accessed using the company code portion of a universal product code, also makes it possible for retailers to readily obtain specific information needed to purchase products directly from manufacturers, establish accounts, identify distributors, and the like. The company information which is made available as contemplated by the invention may be used to automatically establish EDI connections and perform EDI transactions between the servers operated on behalf of retail stores and those operated on behalf of manufacturers.

As illustrated at **875** in FIG. **8**, the cross-referencing server can be used to by an indexing service to provide searchable product and company information indices and

search capabilities. As noted earlier, cross-referencing utility seen at **880** in FIG. **8** may advantageously provide means for accessing the entire contents of its cross-referencing information to a requesting computer, such as a search engine operated by the index service seen at which can then perform conventional "web crawler" indexing of the websites specified by the listed URLs and/or IP addresses, thereby generating complete or partial indexes to all or less than all of the products whose product description locations have been registered. Alternatively, web crawling engines can traverse the links to manufacturers' information found in product listing pages made available by retailers and others.

By storing product and company information in accordance with predetermined schemas, preferably using XML, DTD and RDF techniques as discussed above, indexing services can provide consumers with powerful tools for locating products having selected attributes and for sorting and comparing product based on those attributes. In this way, a consumer can more readily identify particular products which best suit her needs, can view detailed, accurate and up-to-date promotional and specification information on each product directly from the manufacturer, and can then identify the most desirable retail sources for selected products based on price, geography or other criteria. In this way, "bricks and mortar" stores can make their entire existing inventory available for inspection by simply adding a "web register" module to connect their existing bar code checkout and inventory control system with a shared sales server, providing their local customers to "shop at home," make purchases on line, and either pick up or arrange for local delivery of the goods purchased.

It should be understood that the methods and apparatus described above are merely illustrative applications of the principles of the present invention. Numerous modifications may be made by those skilled in the art without departing from the true spirit and scope of the invention.

What is claimed is:

1. The method for disseminating product information via the Internet which comprises, in combination, the steps of:

establishing a cross-referencing resource connected to the Internet which includes a database containing a plurality of cross-references, each of said cross-references specifying the correspondence between a group of one or more universal product code values and the Internet address of a source of information which describes the products designated by said group of one or more code values,

transmitting via the Internet a Web page containing at least one hyperlink including a reference to separately stored information, said reference including a particular universal product code value that uniquely designates a selected product,

employing a Web browser program to receive said Web page and display said Web page to a user,

further employing said Web browser to respond to the activation of said hyperlink by said user by transmitting a first request message to said cross-referencing resource, said first request message containing at least a portion of said particular universal product code value,

processing said first request message at said cross-referencing resource by referring to said database to identify the particular Internet address which corresponds to said particular universal product code value and returning a redirection message to said Web browser which contains said particular Internet address;

employing said Web browser to automatically respond to said redirection message by transmitting a second request message to said particular Internet address;

employing a Web server connected to the Internet at said particular Internet address to respond to said second request message by returning product information describing said selected product to said Web browser, and

employing said Web browser program to automatically display said product information from said Web server to said user.

2. The method set forth in claim **1** wherein said portion is the company code portion of a said particular universal product code and wherein said particular Internet address specifies the location of an Internet resource operated on behalf of the manufacturer specified by said company code portion.

3. The method set forth in claim **1** wherein said information describing the product designated by said particular universal product code is expressed in eXtensible Markup Language.

4. The method set forth in claim **2** wherein said information describing the product designated by said particular universal product code is expressed in eXtensible Markup Language.

5. Apparatus for delivering product information via the Internet which comprises, in combination:

a first computer connected to the Internet for producing a web page containing at least one hypertext link to product information, said hypertext link containing a reference which includes at least a portion of a particular universal product code which designates a particular product,

a second computer connected to the Internet for maintaining a cross-referencing database storing information and operating as a cross-referencing resource which associates universal product codes and corresponding Internet addresses identifying remote resources from which data may be retrieved which describes the products identified by said universal product codes, said second computer including means for responding to a request message containing a least a portion of a given universal product code by transmitting a response message specifying the corresponding Internet address from which information describing the product designed by said universal product code may be retrieved and to which said request message should be redirected, and

a third computer connected to the Internet for executing a web browser program, said web browser program comprising

means for retrieving said web page from said first computer,

means for identifying said hypertext link in said web page and transmitting a first request message to said cross-referencing resource, said request message containing at least a portion of said particular universal product code to obtain from said second computer a response message specifying a particular universal product code to which said first request message should be redirected, and

means automatically responsive to the receipt of said response message for transmitting a second request message to said to said particular Internet address to retrieve product information describing the product designated by said particular universal product code, and

means for receiving and displaying said product information returned to said third computer in response to said second request message.

6. The apparatus set forth in claim 5 wherein said response message takes the form of a hypertext transport protocol response message which includes a location header field containing a destination URL specifying said particular Internet address whereby said request message is redirected to said particular destination URL.

7. The apparatus set forth in claim 5 wherein said database stores a company code portion of a said universal product code and wherein said corresponding Internet address specifies the location of an Internet resource operated on behalf of the entity specified by said company code portion.

8. The apparatus set forth in claim 5 wherein said cross-referencing resource is the Internet Domain Name Service.

9. The apparatus set forth in claim 8 wherein said product information is expressed in extensible markup language.

10. The apparatus set forth in claim 9 wherein said web browser displays said product information in accordance with stylesheet information specified on said web page.

11. The apparatus set forth in claim 10 wherein said product information satisfies the validity requirements contained in a predetermined standard XML Document Type Definition for product information.

12. The apparatus set forth in claim 10 wherein said product information conforms to a schema expressed in accordance with the Resource Description Framework (RDF) and Syntax Specification.

13. The apparatus as set forth in claim 5 wherein said reference in said web page is contained in a hypertext link anchored by a visual element which is retrieved from said second computer by said web browser program when said web page is displayed, said visual element having content which is dependent upon the nature of the information stored in said database relating to said particular universal product code.

14. The apparatus as set forth in claim 13 wherein said visual element is an image file transmitted to said web browser from said second computer in response to an image request message containing at least a portion of said particular universal product code.

* * * * *

US006295513B1

(12) **United States Patent**
Thackston

(10) **Patent No.: US 6,295,513 B1**
(45) **Date of Patent: Sep. 25, 2001**

(54) **NETWORK-BASED SYSTEM FOR THE MANUFACTURE OF PARTS WITH A VIRTUAL COLLABORATIVE ENVIRONMENT FOR DESIGN, DEVELOPEMENT, AND FABRICATOR SELECTION**

(75) Inventor: **James D. Thackston**, Marietta, GA (US)

(73) Assignee: **Eagle Engineering of America, Inc.**, Marietta, GA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/410,619**

(22) Filed: **Oct. 1, 1999**

Related U.S. Application Data

(63) Continuation-in-part of application No. 09/270,007, filed on Mar. 16, 1999, and a continuation-in-part of application No. 09/311,150, filed on May 13, 1999.

(51) **Int. Cl.**[7] **G06F 17/50**

(52) **U.S. Cl.** **703/1**; 703/6; 707/104

(58) **Field of Search** 700/98, 118; 705/1; 707/104, 500; 703/13; 345/442, 440, 421; 716/18

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,964,043	10/1990	Galvin .	
5,278,751	1/1994	Adiano et al. .	
5,321,841	6/1994	East et al. .	
5,444,844	8/1995	Inoue et al. .	
5,644,493	7/1997	Motai et al. .	
5,664,115	9/1997	Fraser .	
5,715,402	2/1998	Popolo .	
5,761,661	6/1998	Coussens et al. .	
5,794,207	8/1998	Walker et al. .	
5,802,502	9/1998	Gell et al. .	
5,815,683	9/1998	Volger .	
5,838,906	11/1998	Doyle et al. .	
5,844,553	12/1998	Hao et al. .	
5,848,427	12/1998	Hyodo .	
5,862,223	1/1999	Walker et al. .	
5,940,082 *	8/1999	Brinegar et al. .	

OTHER PUBLICATIONS

Lisa Kempfer, 'Tools for Web Collaboration', Apr. 1999, Computer Aided Engineering, vol. 18, No. 4, pp 38.*

Pages printed out from Internet on Mar. 15, 1999; "TPN Register," at www.tpnregister.com (13 pages).

Pages printed out from Internet on Mar. 15, 1999; "Understanding Product Data Management" at pdmic/com/undrstnd.htm1#datamgmt (9 pages).

Pages printed out from Internet on Apr. 26, 1999; "GE TPN Post Detailed Service Use Guidelines", at www.tpn.geis-.com/tpn/getting started (18 pages).

Pages printed out from Internet on Apr. 26, 1999; "GE TPN Post Resource Center", at www.tpn.geis.com/tpn/resource center (9 pages).

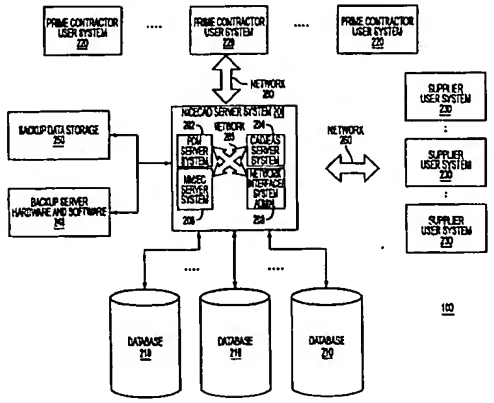
(List continued on next page.)

Primary Examiner—Kevin J. Teska
Assistant Examiner—Russell W. Frejd
(74) *Attorney, Agent, or Firm*—Hunton & Williams

(57) **ABSTRACT**

The invention relates generally to a comprehensive, integrated computer-based system and method for undertaking an engineering design and development effort in a virtual collaborative environment, identifying qualified fabricators for manufacturing a part design based on fabricator capability information stored in a global registry database substantially maintained by the fabricators themselves, and conducting a virtual bidding process whereby electronic representations of three dimensional model and specification data are provided by a central server. The central server further supports the bidding process by providing quasi-real time audio, video and graphics, and the contracts negotiation and formalization steps.

43 Claims, 29 Drawing Sheets



OTHER PUBLICATIONS

Pages printed out from Internet on Apr. 26, 1999; "Thomas-Net Incorporated," at www.thomasnet.com (9 pages).
Pages printed out from Internet on Apr. 26, 1999; "SoluSource," at www.solusource.com (10 pages).
Pages printed out from Internet on Apr. 26, 1999; "Harris InfoSource," at www.harrisinfo.com (12 pages).
Pages printed out from Internet on Apr. 26, 1999; "Muse Technologies," at www.musetech.com (28 pages).
Pages printed out from Internet on Apr. 26, 1999; "Product Data Integration Technologies, Inc., " at www.pdit.com (19 pages).
Pages printed out from Internet on Aug. 9, 1999; "SBA Pro-Net" at www.pro-net.sba.gov (11 pages).
Pages printed out from Internet on Aug. 9, 1999; "FreeMarkets," at www.freemarkets.com (35 pages).
"Process Control," article from *Computer Graphics World,* pp. 69–72 (Aug. 1999).
Pages printed out from Internet on Sep. 20, 1999; "Wecome to Enovia," at www.enovia.com (32 pages).
Pages printed out from Internet on Sep. 20, 1999; "Metaphase," at www.metaphasetech.com (24 pages).
Pages printed out from Internet on Sep. 20, 1999; "Bentleys Project Bank . . . , " at www.bentley.com (85 pages).
Pages printed out from Internet on Sep. 20, 1999; "Unigraphics Solutions," at www.ugsolutions.com (13 pages).
Pages printed out from Internet on Sep. 20, 1999; "Parametric Technology Corporation Features," at www.ptc.com (20 pages).
Pages printed out from Internet on Sep. 20, 1999; "Open Virtual Factory™," at www.eai.com (57 pages).
Pages printed out from Internet on Oct. 18, 1999; "About TradeMatrix," at www.tradematrix.com (46 pages).
Pages printed out from Internet "webcast" on Oct. 26, 1999; "Collaborative Product Commerce," at webevents.broadcast.com (11 pages).
Pages printed out from Internet on Mar. 20, 2000; "CoCreate," at www.cocreate (25 pages).
Pages printed out from Internet on Mar. 20, 1999; "Alibre.com," at www.alibre.com (9 pages).
Pages printed out from Internet on Mar. 20, 1999; "SupplierMarket.com," at www.suppliermarket.com (13 pages).
Lisa Kempfer, 'Tools for Web Collaboration', Apr. 1999, Computer-Aided Engineering, Apr. 1999, pp38.*

* cited by examiner

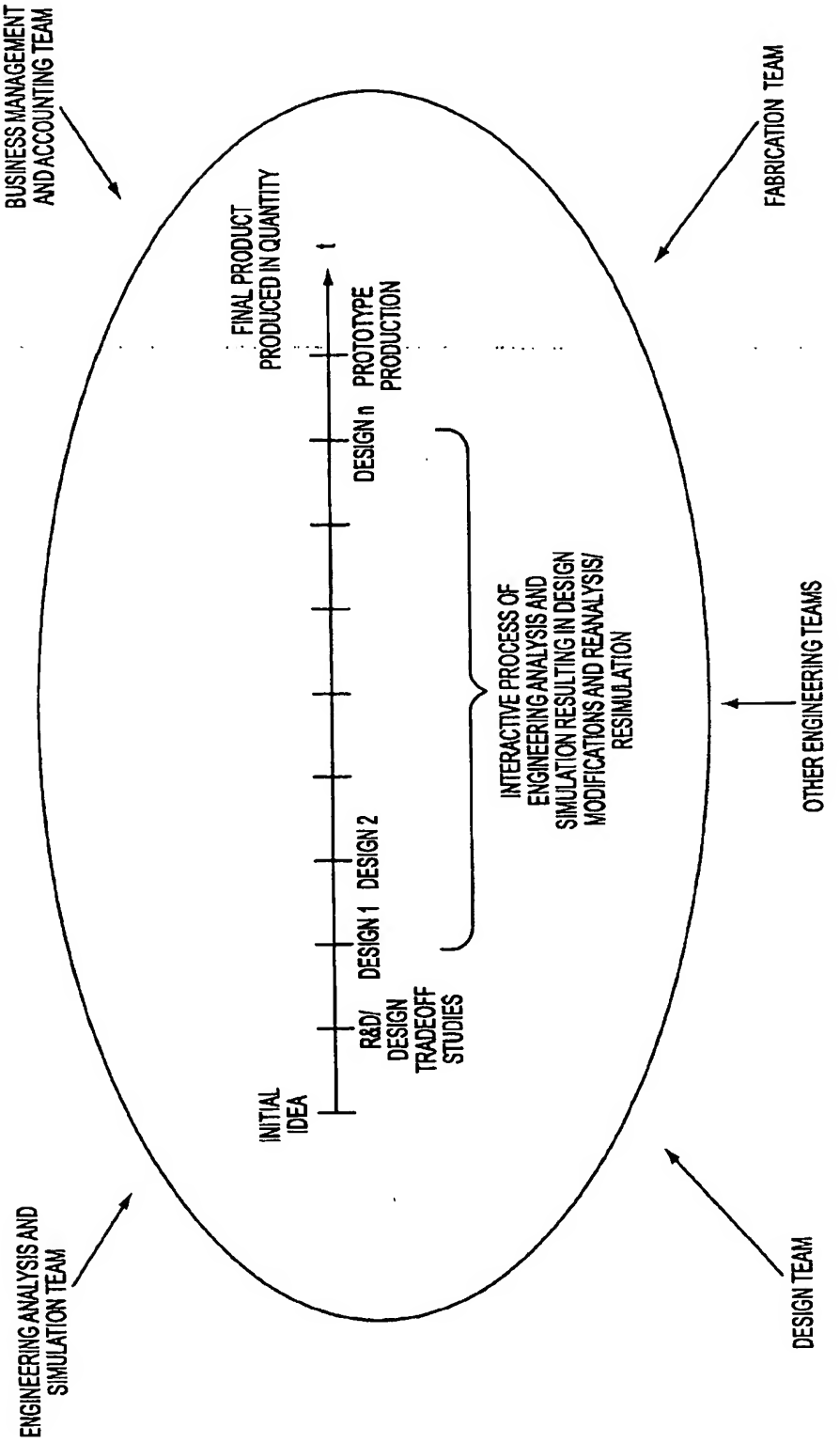

BUSINESS MANAGEMENT AND ACCOUNTING TEAM
FABRICATION TEAM
FINAL PRODUCT PRODUCED IN QUANTITY
t
PROTOTYPE PRODUCTION
DESIGN n
INTERACTIVE PROCESS OF ENGINEERING ANALYSIS AND SIMULATION RESULTING IN DESIGN MODIFICATIONS AND REANALYSIS/ RESIMULATION
DESIGN 2
DESIGN 1
R&D/ DESIGN TRADEOFF STUDIES
INITIAL IDEA
OTHER ENGINEERING TEAMS
ENGINEERING ANALYSIS AND SIMULATION TEAM
DESIGN TEAM

FIG. 1

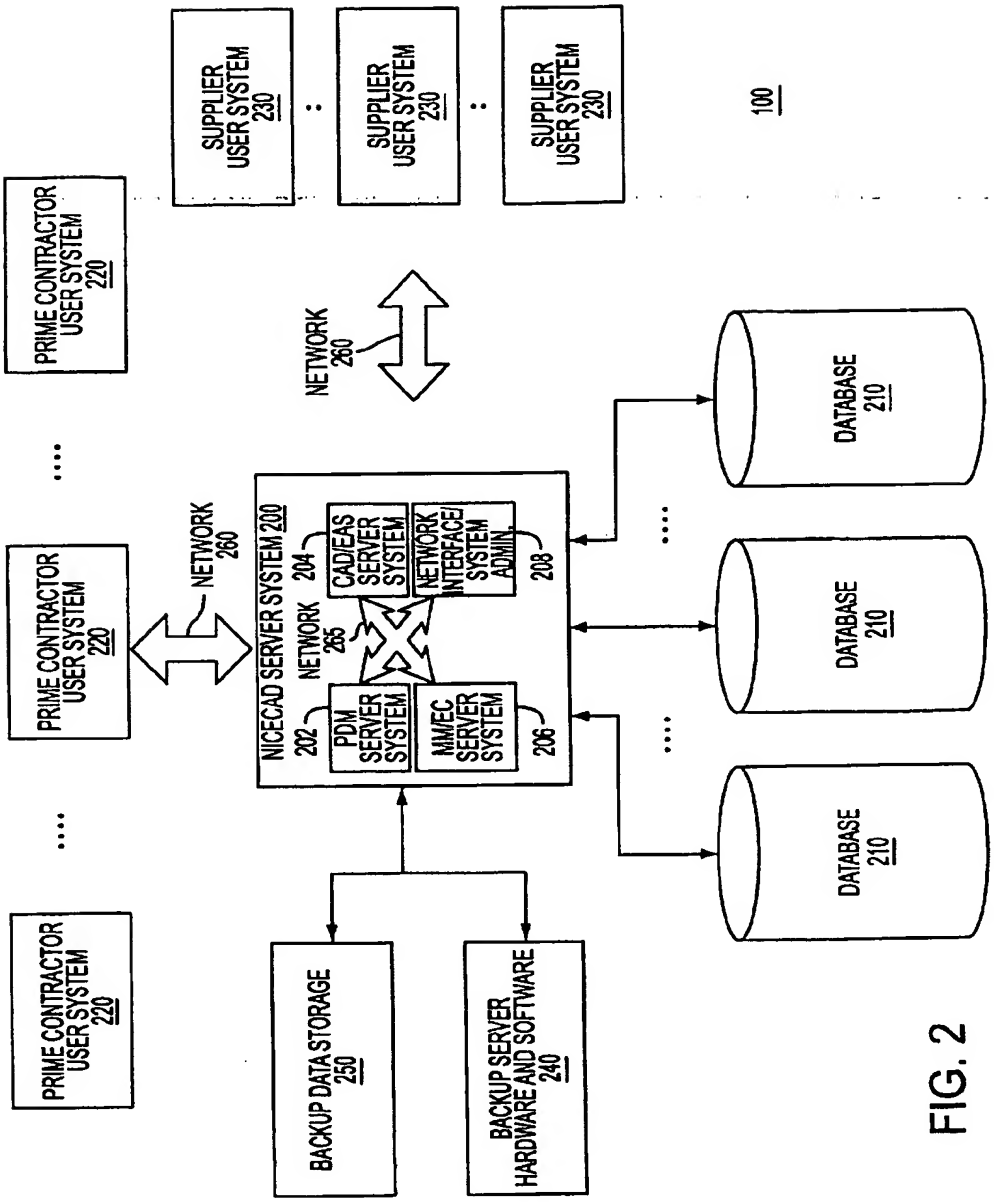

SUPPLIER USER SYSTEM 230
SUPPLIER USER SYSTEM 230
SUPPLIER USER SYSTEM 230
100
PRIME CONTRACTOR USER SYSTEM 220
NETWORK 260
DATABASE 210
DATABASE 210
DATABASE 210
PRIME CONTRACTOR USER SYSTEM 220
NETWORK 260
NICECAD SERVER SYSTEM 200
CAD/EAS SERVER SYSTEM 204
NETWORK INTERFACE/ SYSTEM ADMIN. 208
NETWORK 265
PDM SERVER SYSTEM 202
MM/EC SERVER SYSTEM 206
PRIME CONTRACTOR USER SYSTEM 220
BACKUP DATA STORAGE 250
BACKUP SERVER HARDWARE AND SOFTWARE 240

FIG. 2

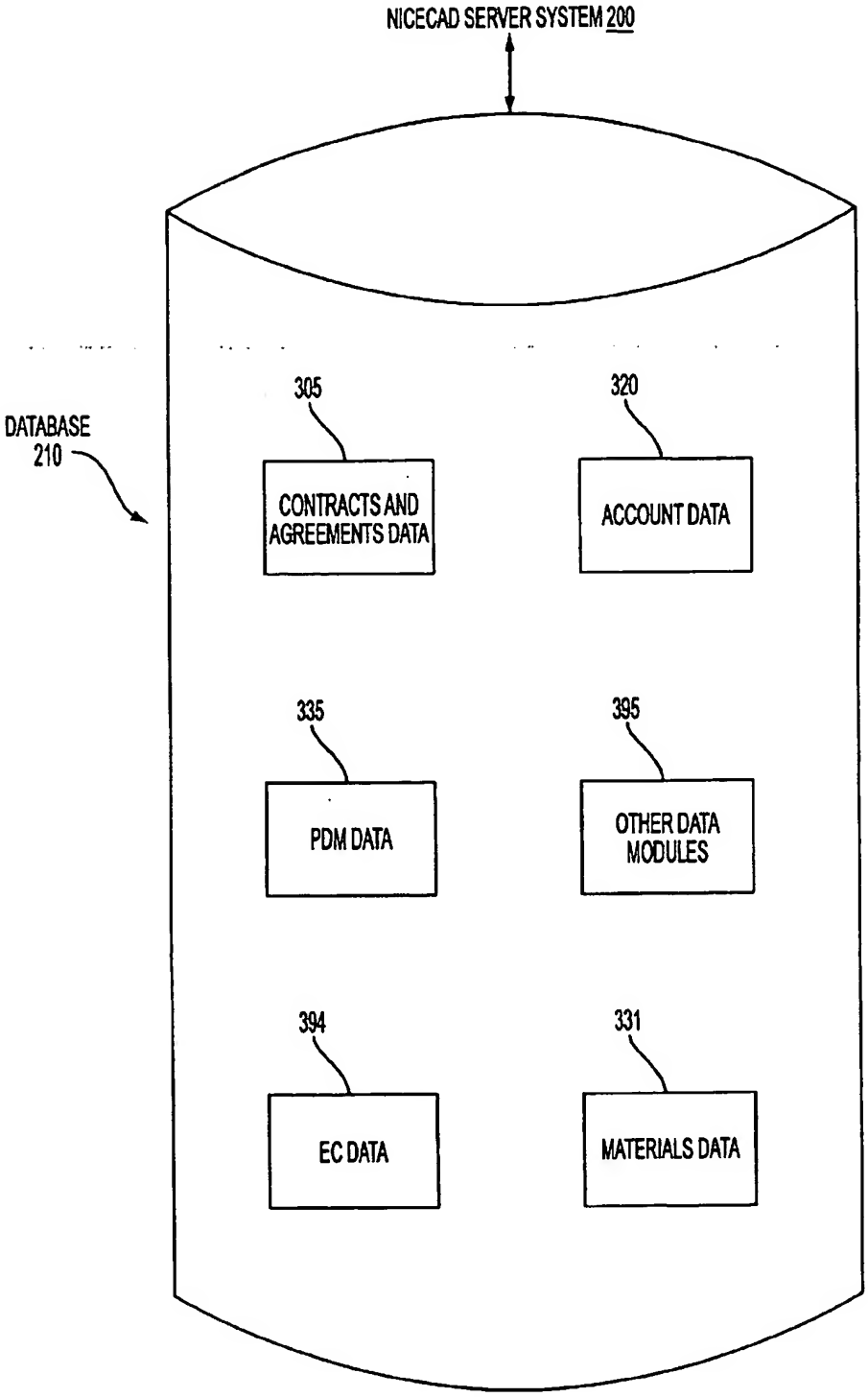

NICECAD SERVER SYSTEM 200
DATABASE 210
305
CONTRACTS AND AGREEMENTS DATA
320
ACCOUNT DATA
335
PDM DATA
395
OTHER DATA MODULES
394
EC DATA
331
MATERIALS DATA

FIG. 3

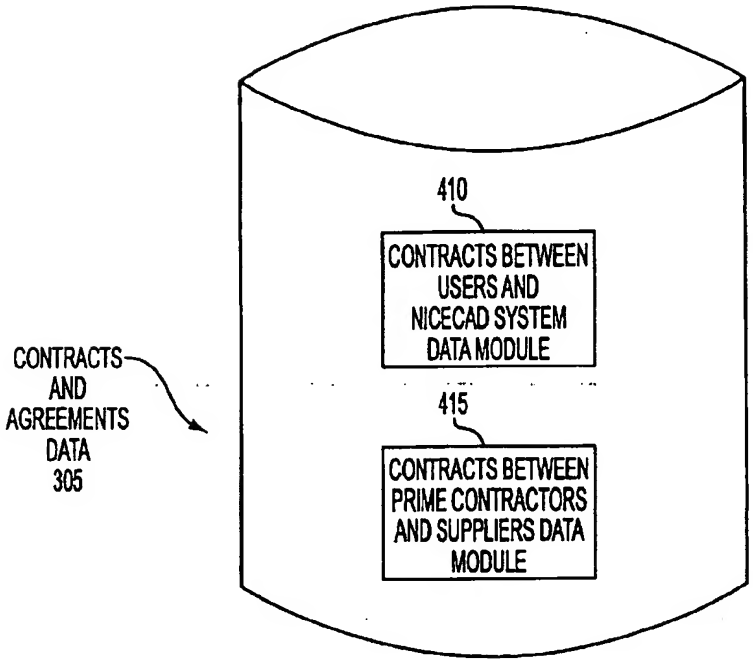

410
CONTRACTS BETWEEN USERS AND NICECAD SYSTEM DATA MODULE
415
CONTRACTS BETWEEN PRIME CONTRACTORS AND SUPPLIERS DATA MODULE
CONTRACTS AND AGREEMENTS DATA 305

FIG. 4

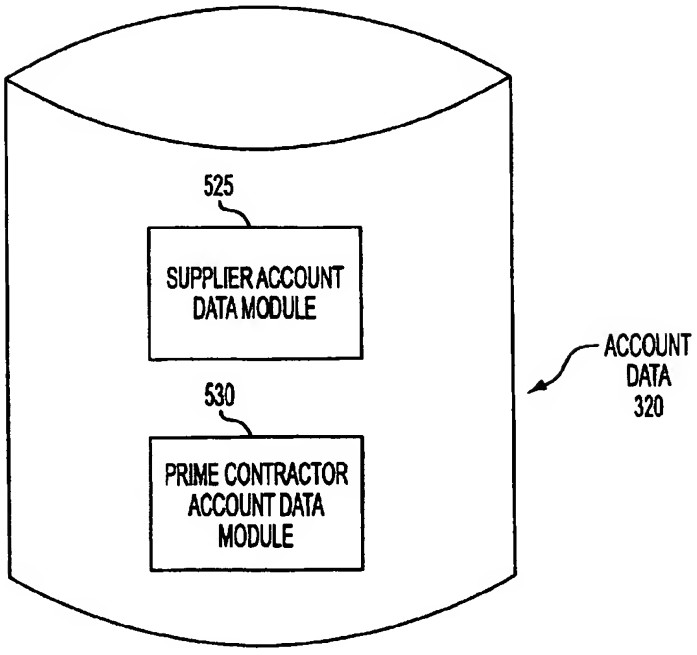

525
SUPPLIER ACCOUNT DATA MODULE
530
PRIME CONTRACTOR ACCOUNT DATA MODULE
ACCOUNT DATA 320

FIG. 5

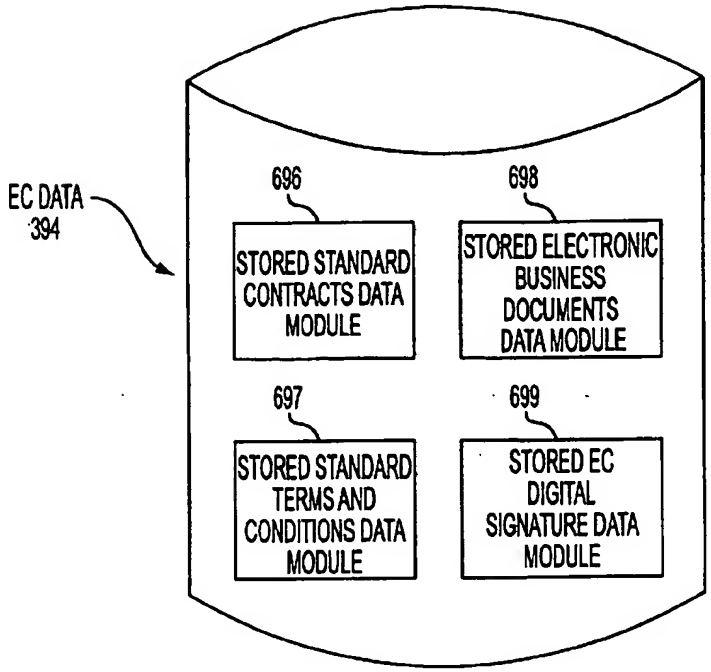

EC DATA
394
696
STORED STANDARD CONTRACTS DATA MODULE
698
STORED ELECTRONIC BUSINESS DOCUMENTS DATA MODULE
697
STORED STANDARD TERMS AND CONDITIONS DATA MODULE
699
STORED EC DIGITAL SIGNATURE DATA MODULE

FIG. 6

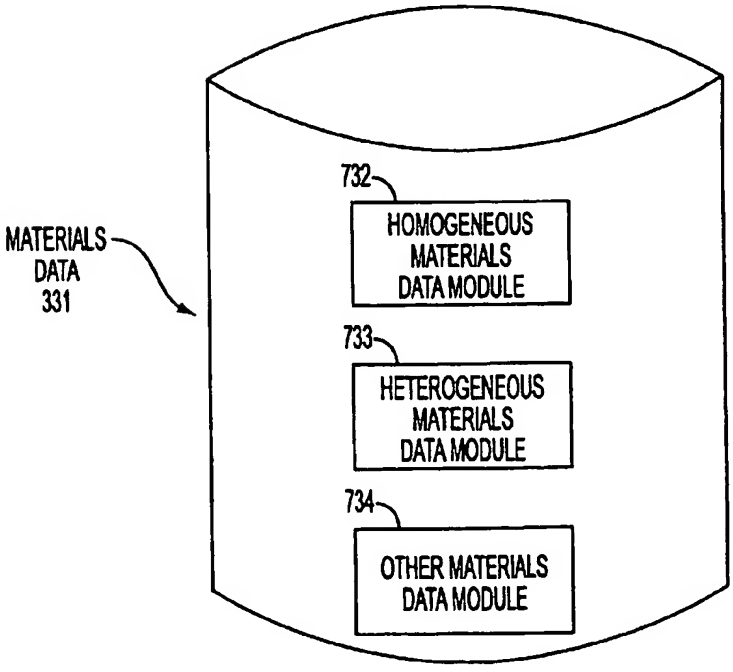

MATERIALS DATA
331
732
HOMOGENEOUS MATERIALS DATA MODULE
733
HETEROGENEOUS MATERIALS DATA MODULE
734
OTHER MATERIALS DATA MODULE

FIG. 7

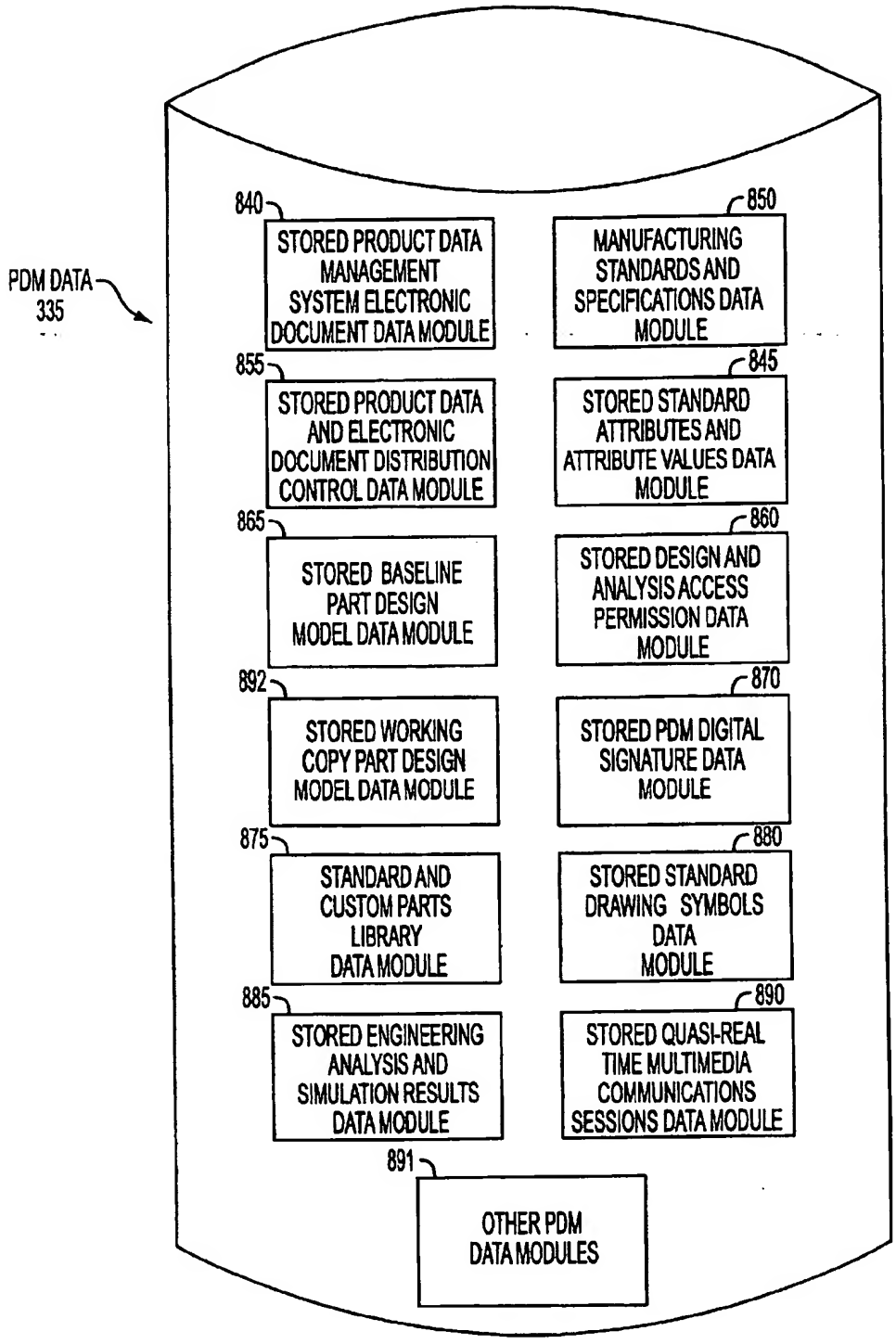

PDM DATA
335
840
STORED PRODUCT DATA MANAGEMENT SYSTEM ELECTRONIC DOCUMENT DATA MODULE
850
MANUFACTURING STANDARDS AND SPECIFICATIONS DATA MODULE
855
STORED PRODUCT DATA AND ELECTRONIC DOCUMENT DISTRIBUTION CONTROL DATA MODULE
845
STORED STANDARD ATTRIBUTES AND ATTRIBUTE VALUES DATA MODULE
865
STORED BASELINE PART DESIGN MODEL DATA MODULE
860
STORED DESIGN AND ANALYSIS ACCESS PERMISSION DATA MODULE
892
STORED WORKING COPY PART DESIGN MODEL DATA MODULE
870
STORED PDM DIGITAL SIGNATURE DATA MODULE
875
STANDARD AND CUSTOM PARTS LIBRARY DATA MODULE
880
STORED STANDARD DRAWING SYMBOLS DATA MODULE
885
STORED ENGINEERING ANALYSIS AND SIMULATION RESULTS DATA MODULE
890
STORED QUASI-REAL TIME MULTIMEDIA COMMUNICATIONS SESSIONS DATA MODULE
891
OTHER PDM DATA MODULES

FIG. 8

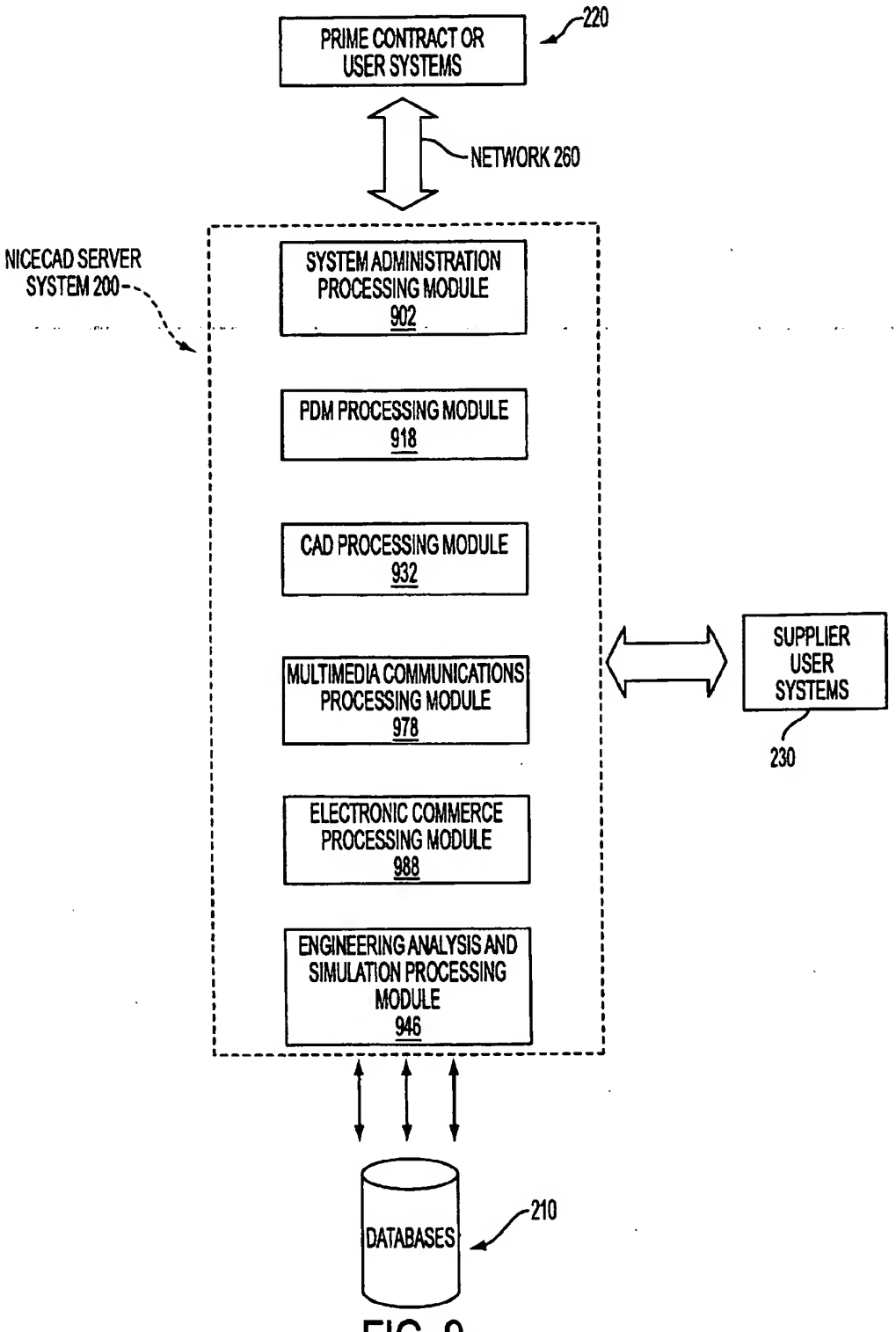

PRIME CONTRACT OR USER SYSTEMS
220
NETWORK 260
NICECAD SERVER SYSTEM 200
SYSTEM ADMINISTRATION PROCESSING MODULE 902
PDM PROCESSING MODULE 918
CAD PROCESSING MODULE 932
MULTIMEDIA COMMUNICATIONS PROCESSING MODULE 978
ELECTRONIC COMMERCE PROCESSING MODULE 988
ENGINEERING ANALYSIS AND SIMULATION PROCESSING MODULE 946
SUPPLIER USER SYSTEMS
230
DATABASES
210

FIG. 9

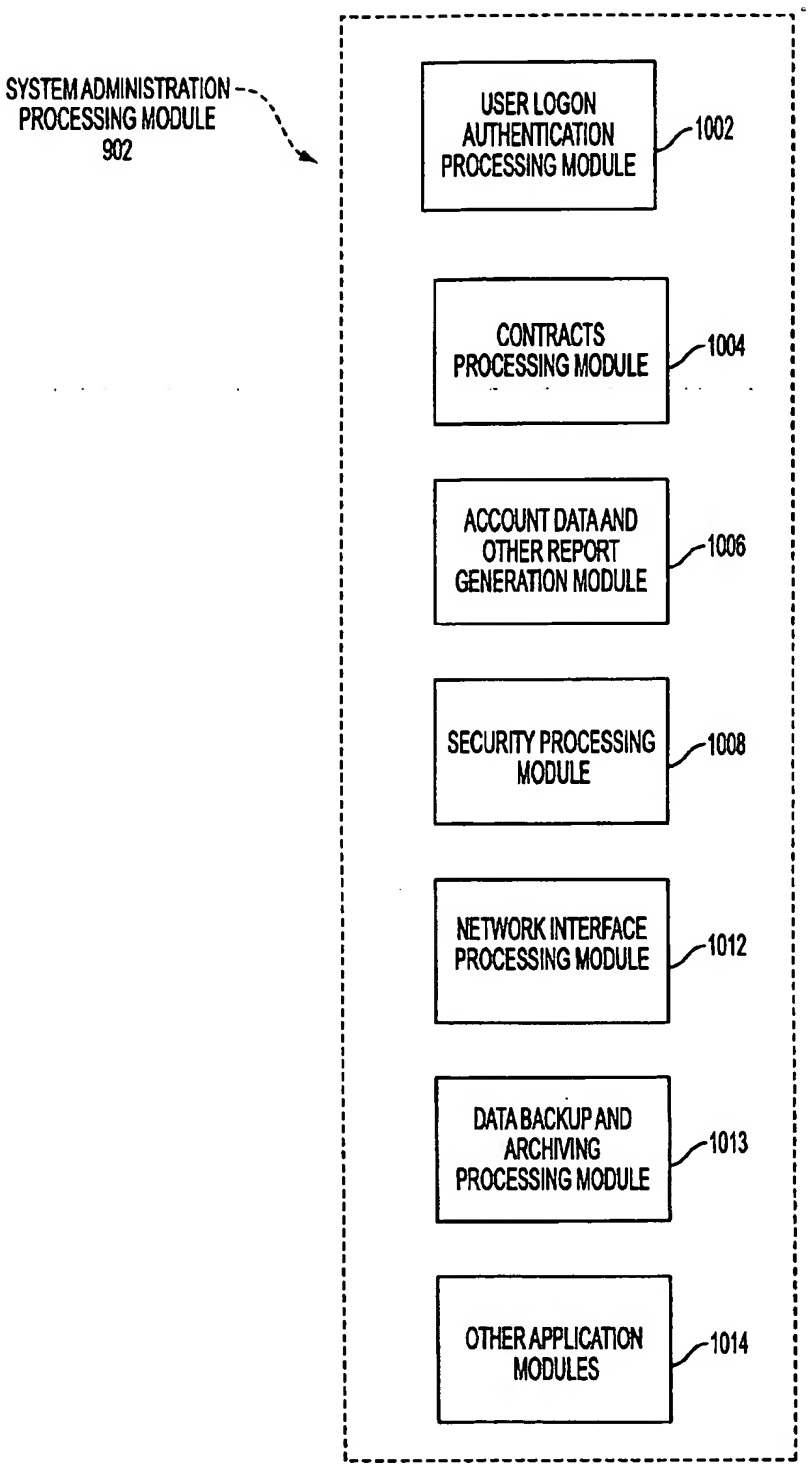

SYSTEM ADMINISTRATION PROCESSING MODULE 902
USER LOGON AUTHENTICATION PROCESSING MODULE
1002
CONTRACTS PROCESSING MODULE
1004
ACCOUNT DATA AND OTHER REPORT GENERATION MODULE
1006
SECURITY PROCESSING MODULE
1008
NETWORK INTERFACE PROCESSING MODULE
1012
DATA BACKUP AND ARCHIVING PROCESSING MODULE
1013
OTHER APPLICATION MODULES
1014

FIG. 10

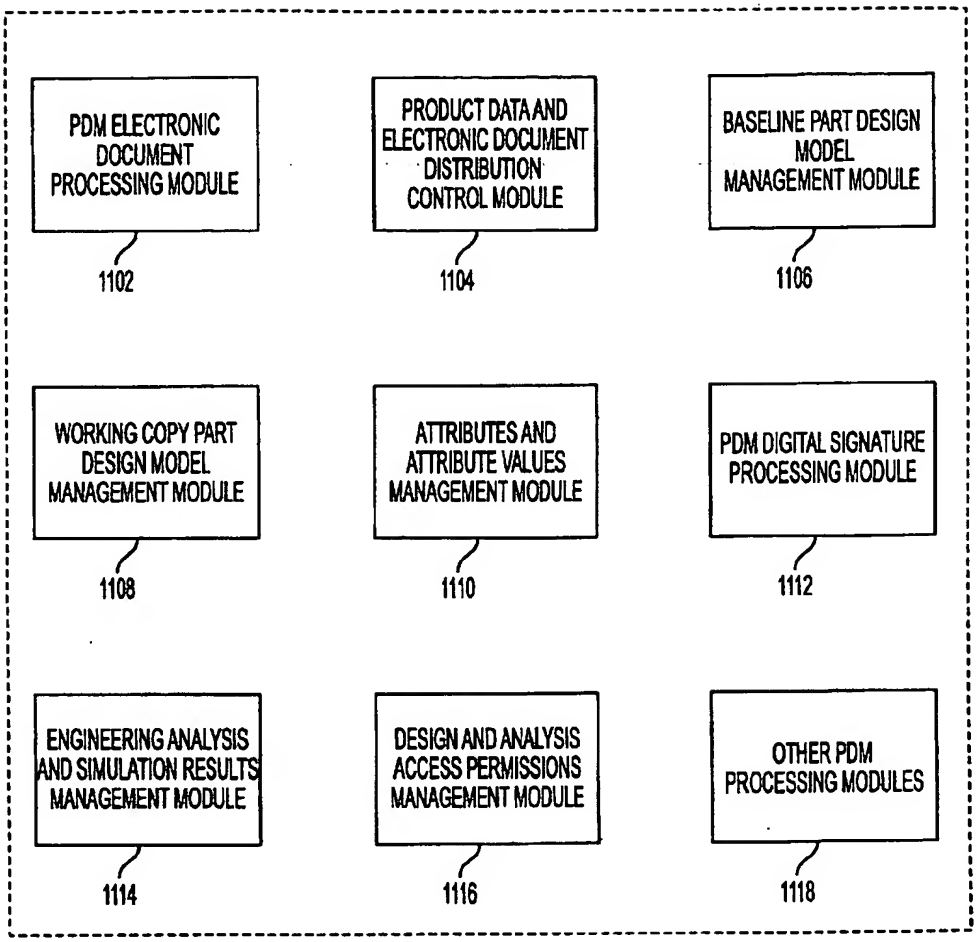

PDM ELECTRONIC DOCUMENT PROCESSING MODULE
1102
PRODUCT DATA AND ELECTRONIC DOCUMENT DISTRIBUTION CONTROL MODULE
1104
BASELINE PART DESIGN MODEL MANAGEMENT MODULE
1106
WORKING COPY PART DESIGN MODEL MANAGEMENT MODULE
1108
ATTRIBUTES AND ATTRIBUTE VALUES MANAGEMENT MODULE
1110
PDM DIGITAL SIGNATURE PROCESSING MODULE
1112
ENGINEERING ANALYSIS AND SIMULATION RESULTS MANAGEMENT MODULE
1114
DESIGN AND ANALYSIS ACCESS PERMISSIONS MANAGEMENT MODULE
1116
OTHER PDM PROCESSING MODULES
1118
PDM PROCESSING MODULE
918

FIG. 11

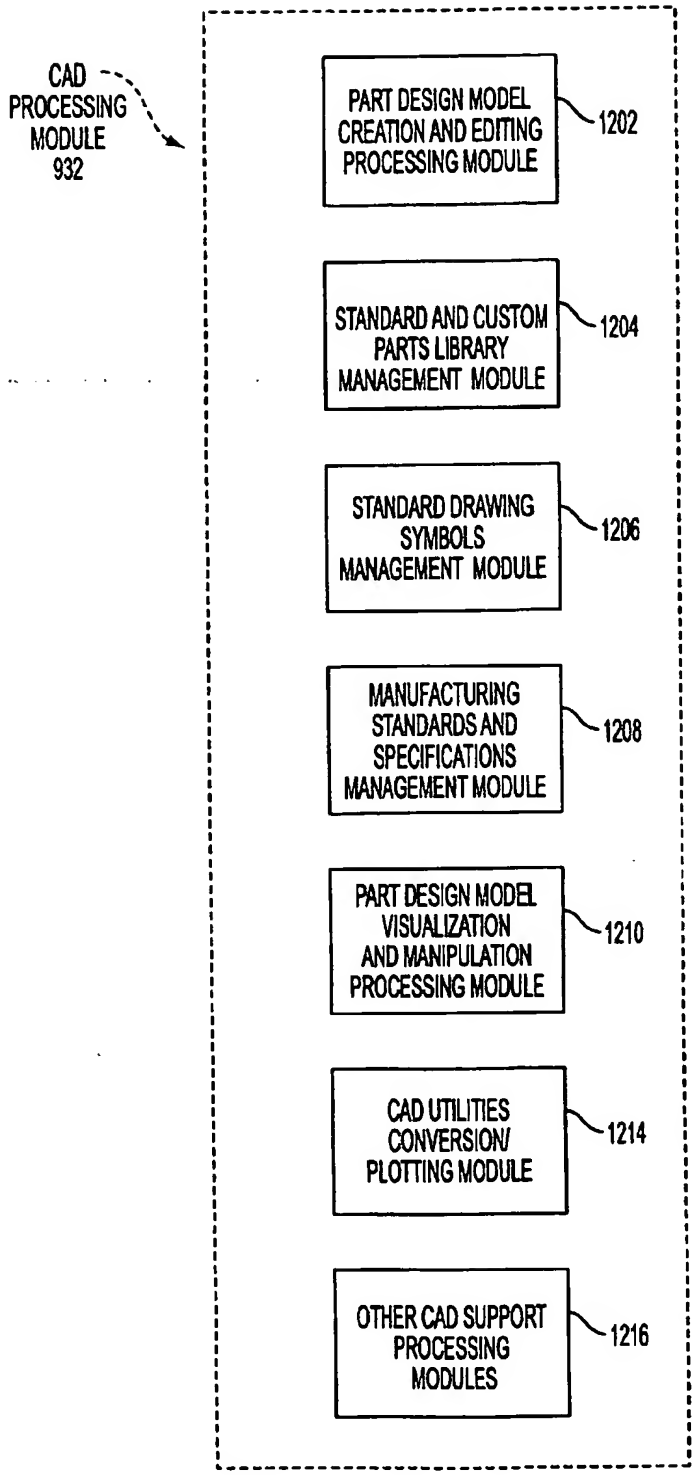

CAD PROCESSING MODULE 932
PART DESIGN MODEL CREATION AND EDITING PROCESSING MODULE
1202
STANDARD AND CUSTOM PARTS LIBRARY MANAGEMENT MODULE
1204
STANDARD DRAWING SYMBOLS MANAGEMENT MODULE
1206
MANUFACTURING STANDARDS AND SPECIFICATIONS MANAGEMENT MODULE
1208
PART DESIGN MODEL VISUALIZATION AND MANIPULATION PROCESSING MODULE
1210
CAD UTILITIES CONVERSION/ PLOTTING MODULE
1214
OTHER CAD SUPPORT PROCESSING MODULES
1216

FIG. 12

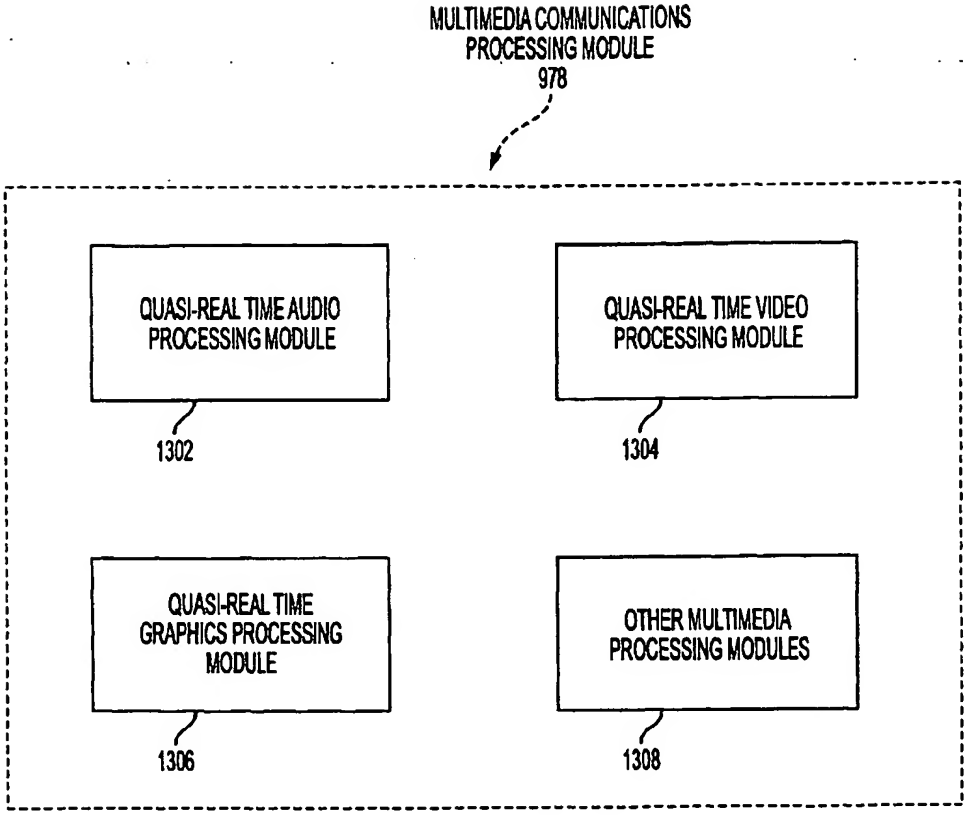

MULTIMEDIA COMMUNICATIONS PROCESSING MODULE
978
QUASI-REAL TIME AUDIO PROCESSING MODULE
1302
QUASI-REAL TIME VIDEO PROCESSING MODULE
1304
QUASI-REAL TIME GRAPHICS PROCESSING MODULE
1306
OTHER MULTIMEDIA PROCESSING MODULES
1308

FIG. 13

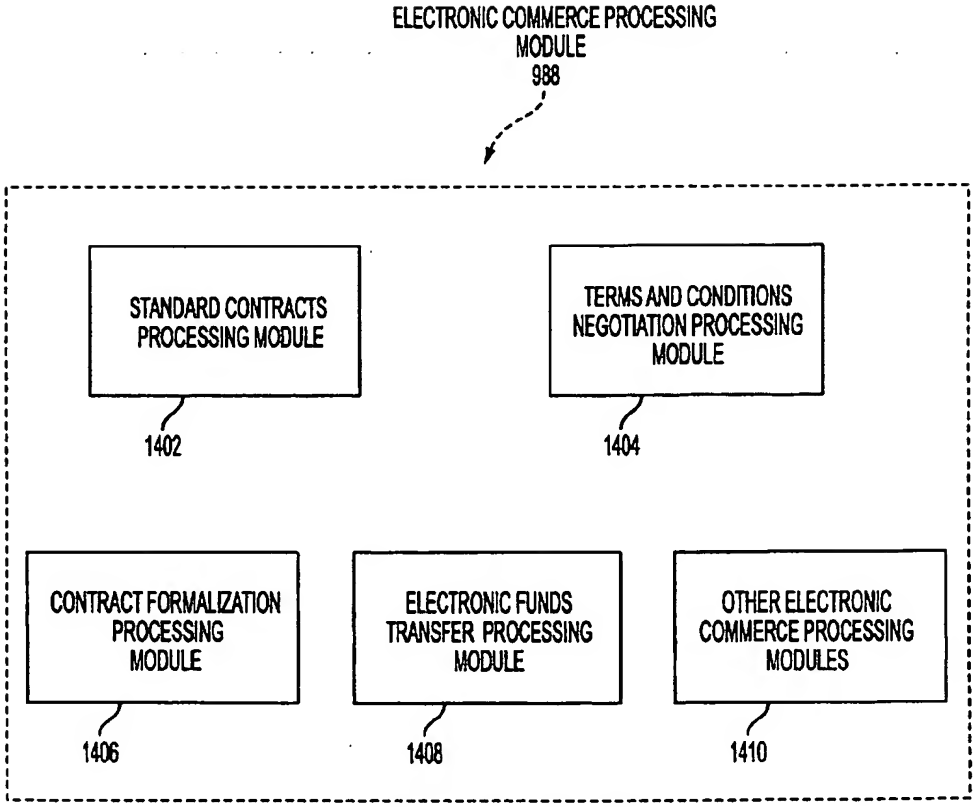

ELECTRONIC COMMERCE PROCESSING MODULE
988
STANDARD CONTRACTS PROCESSING MODULE
1402
TERMS AND CONDITIONS NEGOTIATION PROCESSING MODULE
1404
CONTRACT FORMALIZATION PROCESSING MODULE
1406
ELECTRONIC FUNDS TRANSFER PROCESSING MODULE
1408
OTHER ELECTRONIC COMMERCE PROCESSING MODULES
1410

FIG. 14

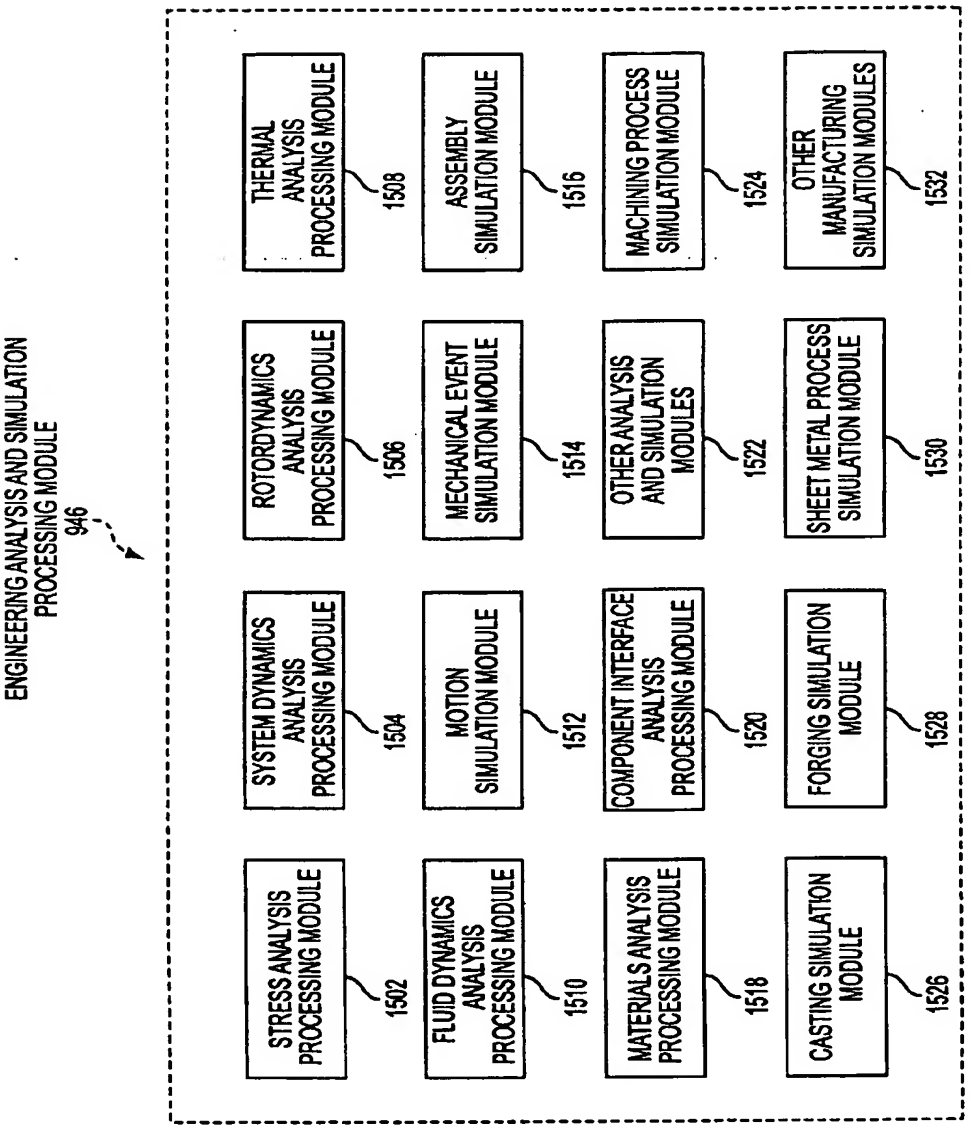

ENGINEERING ANALYSIS AND SIMULATION PROCESSING MODULE
946
STRESS ANALYSIS PROCESSING MODULE
1502
SYSTEM DYNAMICS ANALYSIS PROCESSING MODULE
1504
ROTORDYNAMICS ANALYSIS PROCESSING MODULE
1506
THERMAL ANALYSIS PROCESSING MODULE
1508
FLUID DYNAMICS ANALYSIS PROCESSING MODULE
1510
MOTION SIMULATION MODULE
1512
MECHANICAL EVENT SIMULATION MODULE
1514
ASSEMBLY SIMULATION MODULE
1516
MATERIALS ANALYSIS PROCESSING MODULE
1518
COMPONENT INTERFACE ANALYSIS PROCESSING MODULE
1520
OTHER ANALYSIS AND SIMULATION MODULES
1522
MACHINING PROCESS SIMULATION MODULE
1524
CASTING SIMULATION MODULE
1526
FORGING SIMULATION MODULE
1528
SHEET METAL PROCESS SIMULATION MODULE
1530
OTHER MANUFACTURING SIMULATION MODULES
1532

FIG. 15

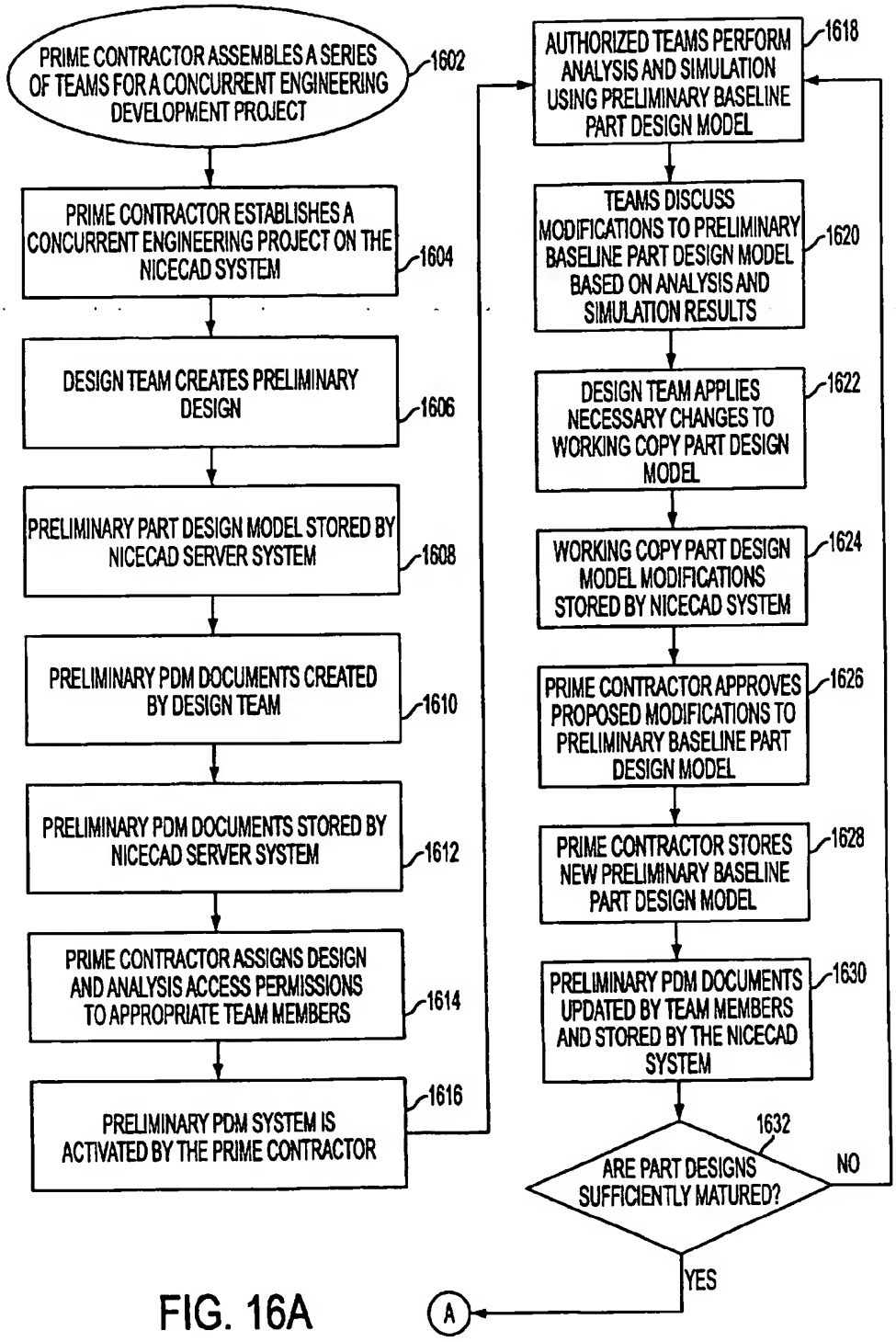

PRIME CONTRACTOR ASSEMBLES A SERIES OF TEAMS FOR A CONCURRENT ENGINEERING DEVELOPMENT PROJECT
1602
PRIME CONTRACTOR ESTABLISHES A CONCURRENT ENGINEERING PROJECT ON THE NICECAD SYSTEM
1604
DESIGN TEAM CREATES PRELIMINARY DESIGN
1606
PRELIMINARY PART DESIGN MODEL STORED BY NICECAD SERVER SYSTEM
1608
PRELIMINARY PDM DOCUMENTS CREATED BY DESIGN TEAM
1610
PRELIMINARY PDM DOCUMENTS STORED BY NICECAD SERVER SYSTEM
1612
PRIME CONTRACTOR ASSIGNS DESIGN AND ANALYSIS ACCESS PERMISSIONS TO APPROPRIATE TEAM MEMBERS
1614
PRELIMINARY PDM SYSTEM IS ACTIVATED BY THE PRIME CONTRACTOR
1616
AUTHORIZED TEAMS PERFORM ANALYSIS AND SIMULATION USING PRELIMINARY BASELINE PART DESIGN MODEL
1618
TEAMS DISCUSS MODIFICATIONS TO PRELIMINARY BASELINE PART DESIGN MODEL BASED ON ANALYSIS AND SIMULATION RESULTS
1620
DESIGN TEAM APPLIES NECESSARY CHANGES TO WORKING COPY PART DESIGN MODEL
1622
WORKING COPY PART DESIGN MODEL MODIFICATIONS STORED BY NICECAD SYSTEM
1624
PRIME CONTRACTOR APPROVES PROPOSED MODIFICATIONS TO PRELIMINARY BASELINE PART DESIGN MODEL
1626
PRIME CONTRACTOR STORES NEW PRELIMINARY BASELINE PART DESIGN MODEL
1628
PRELIMINARY PDM DOCUMENTS UPDATED BY TEAM MEMBERS AND STORED BY THE NICECAD SYSTEM
1630
1632
ARE PART DESIGNS SUFFICIENTLY MATURED?
NO
YES
A

FIG. 16A

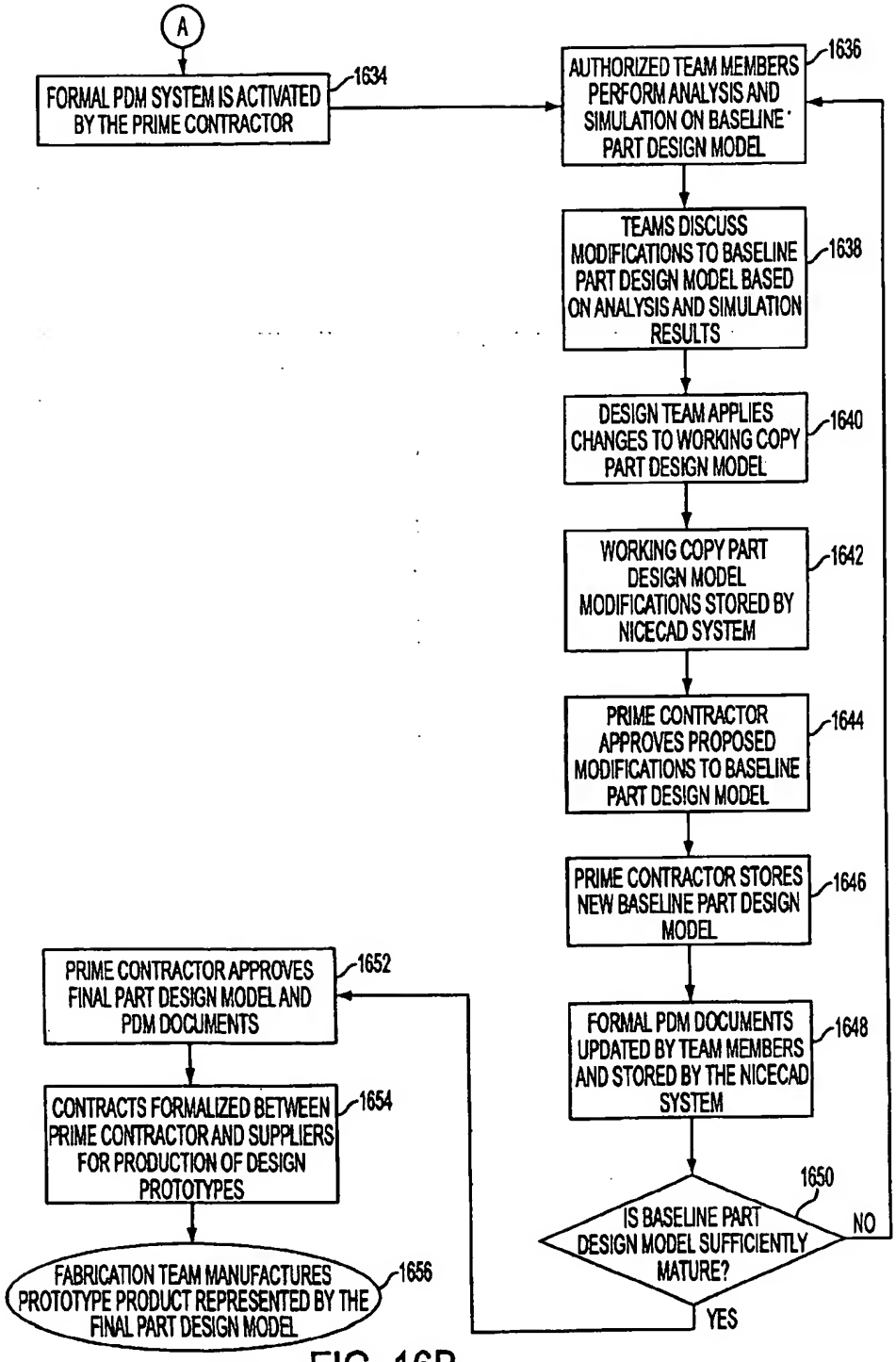

A
FORMAL PDM SYSTEM IS ACTIVATED BY THE PRIME CONTRACTOR
1634
AUTHORIZED TEAM MEMBERS PERFORM ANALYSIS AND SIMULATION ON BASELINE PART DESIGN MODEL
1636
TEAMS DISCUSS MODIFICATIONS TO BASELINE PART DESIGN MODEL BASED ON ANALYSIS AND SIMULATION RESULTS
1638
DESIGN TEAM APPLIES CHANGES TO WORKING COPY PART DESIGN MODEL
1640
WORKING COPY PART DESIGN MODEL MODIFICATIONS STORED BY NICECAD SYSTEM
1642
PRIME CONTRACTOR APPROVES PROPOSED MODIFICATIONS TO BASELINE PART DESIGN MODEL
1644
PRIME CONTRACTOR STORES NEW BASELINE PART DESIGN MODEL
1646
FORMAL PDM DOCUMENTS UPDATED BY TEAM MEMBERS AND STORED BY THE NICECAD SYSTEM
1648
1650
IS BASELINE PART DESIGN MODEL SUFFICIENTLY MATURE?
NO
YES
PRIME CONTRACTOR APPROVES FINAL PART DESIGN MODEL AND PDM DOCUMENTS
1652
CONTRACTS FORMALIZED BETWEEN PRIME CONTRACTOR AND SUPPLIERS FOR PRODUCTION OF DESIGN PROTOTYPES
1654
FABRICATION TEAM MANUFACTURES PROTOTYPE PRODUCT REPRESENTED BY THE FINAL PART DESIGN MODEL
1656

FIG. 16B

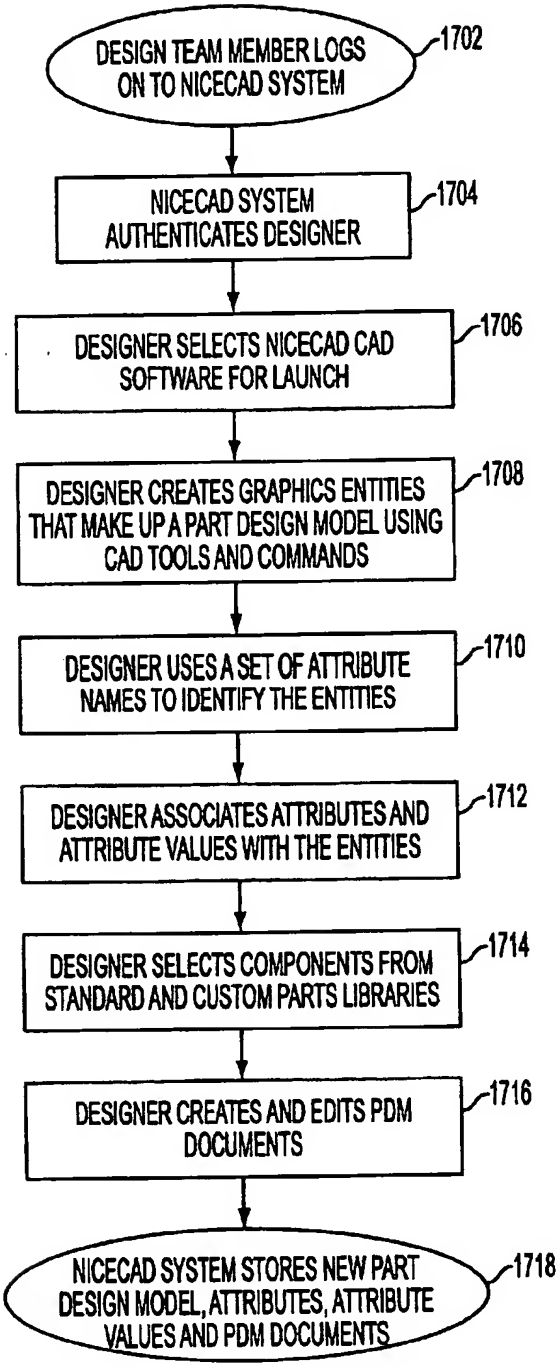

DESIGN TEAM MEMBER LOGS ON TO NICECAD SYSTEM
1702
NICECAD SYSTEM AUTHENTICATES DESIGNER
1704
DESIGNER SELECTS NICECAD CAD SOFTWARE FOR LAUNCH
1706
DESIGNER CREATES GRAPHICS ENTITIES THAT MAKE UP A PART DESIGN MODEL USING CAD TOOLS AND COMMANDS
1708
DESIGNER USES A SET OF ATTRIBUTE NAMES TO IDENTIFY THE ENTITIES
1710
DESIGNER ASSOCIATES ATTRIBUTES AND ATTRIBUTE VALUES WITH THE ENTITIES
1712
DESIGNER SELECTS COMPONENTS FROM STANDARD AND CUSTOM PARTS LIBRARIES
1714
DESIGNER CREATES AND EDITS PDM DOCUMENTS
1716
NICECAD SYSTEM STORES NEW PART DESIGN MODEL, ATTRIBUTES, ATTRIBUTE VALUES AND PDM DOCUMENTS
1718

FIG. 17

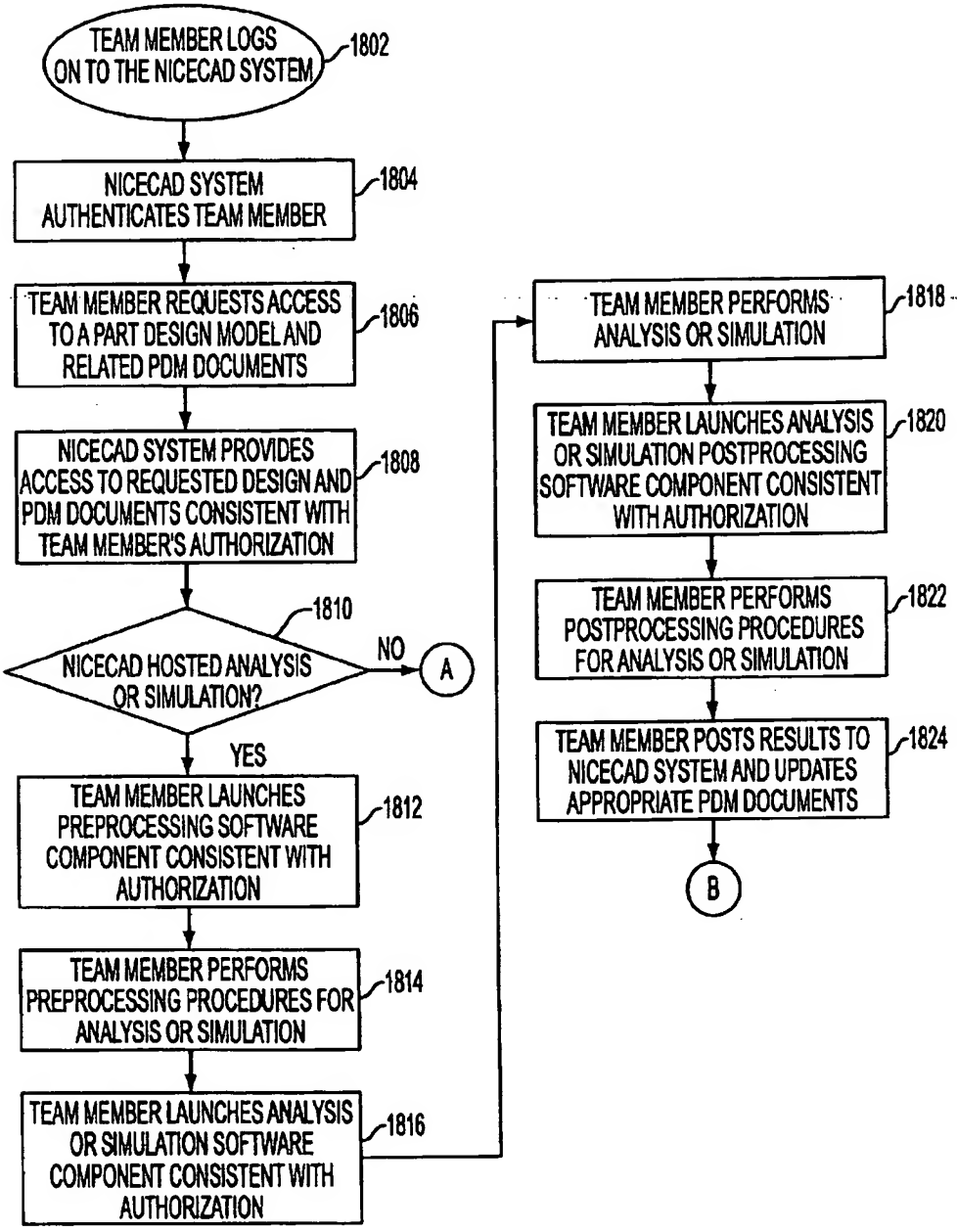

TEAM MEMBER LOGS ON TO THE NICECAD SYSTEM
1802
NICECAD SYSTEM AUTHENTICATES TEAM MEMBER
1804
TEAM MEMBER REQUESTS ACCESS TO A PART DESIGN MODEL AND RELATED PDM DOCUMENTS
1806
NICECAD SYSTEM PROVIDES ACCESS TO REQUESTED DESIGN AND PDM DOCUMENTS CONSISTENT WITH TEAM MEMBER'S AUTHORIZATION
1808
1810
NICECAD HOSTED ANALYSIS OR SIMULATION?
NO
A
YES
TEAM MEMBER LAUNCHES PREPROCESSING SOFTWARE COMPONENT CONSISTENT WITH AUTHORIZATION
1812
TEAM MEMBER PERFORMS PREPROCESSING PROCEDURES FOR ANALYSIS OR SIMULATION
1814
TEAM MEMBER LAUNCHES ANALYSIS OR SIMULATION SOFTWARE COMPONENT CONSISTENT WITH AUTHORIZATION
1816
TEAM MEMBER PERFORMS ANALYSIS OR SIMULATION
1818
TEAM MEMBER LAUNCHES ANALYSIS OR SIMULATION POSTPROCESSING SOFTWARE COMPONENT CONSISTENT WITH AUTHORIZATION
1820
TEAM MEMBER PERFORMS POSTPROCESSING PROCEDURES FOR ANALYSIS OR SIMULATION
1822
TEAM MEMBER POSTS RESULTS TO NICECAD SYSTEM AND UPDATES APPROPRIATE PDM DOCUMENTS
1824
B

FIG. 18A

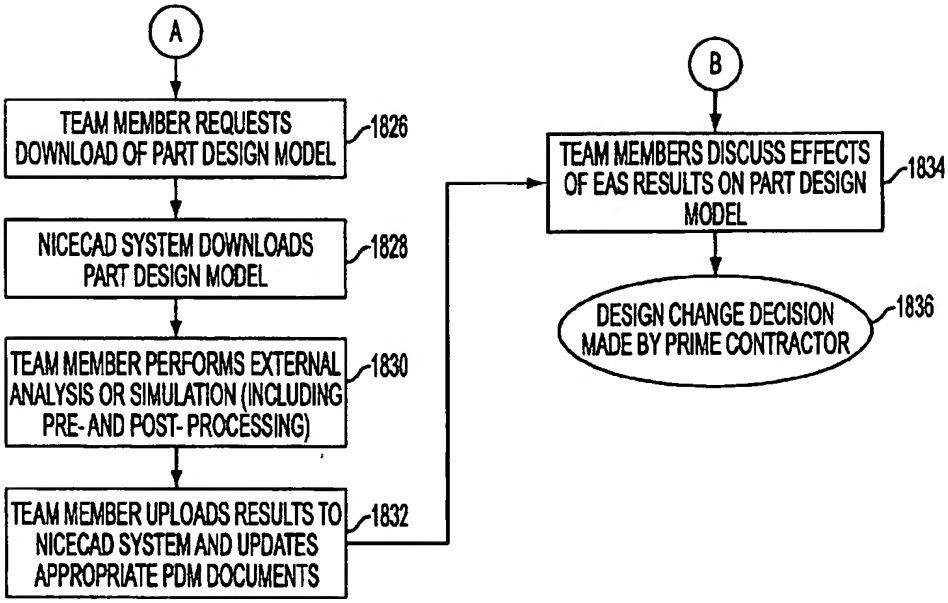

A
TEAM MEMBER REQUESTS DOWNLOAD OF PART DESIGN MODEL
1826
NICECAD SYSTEM DOWNLOADS PART DESIGN MODEL
1828
TEAM MEMBER PERFORMS EXTERNAL ANALYSIS OR SIMULATION (INCLUDING PRE- AND POST- PROCESSING)
1830
TEAM MEMBER UPLOADS RESULTS TO NICECAD SYSTEM AND UPDATES APPROPRIATE PDM DOCUMENTS
1832
B
TEAM MEMBERS DISCUSS EFFECTS OF EAS RESULTS ON PART DESIGN MODEL
1834
DESIGN CHANGE DECISION MADE BY PRIME CONTRACTOR
1836

FIG. 18B

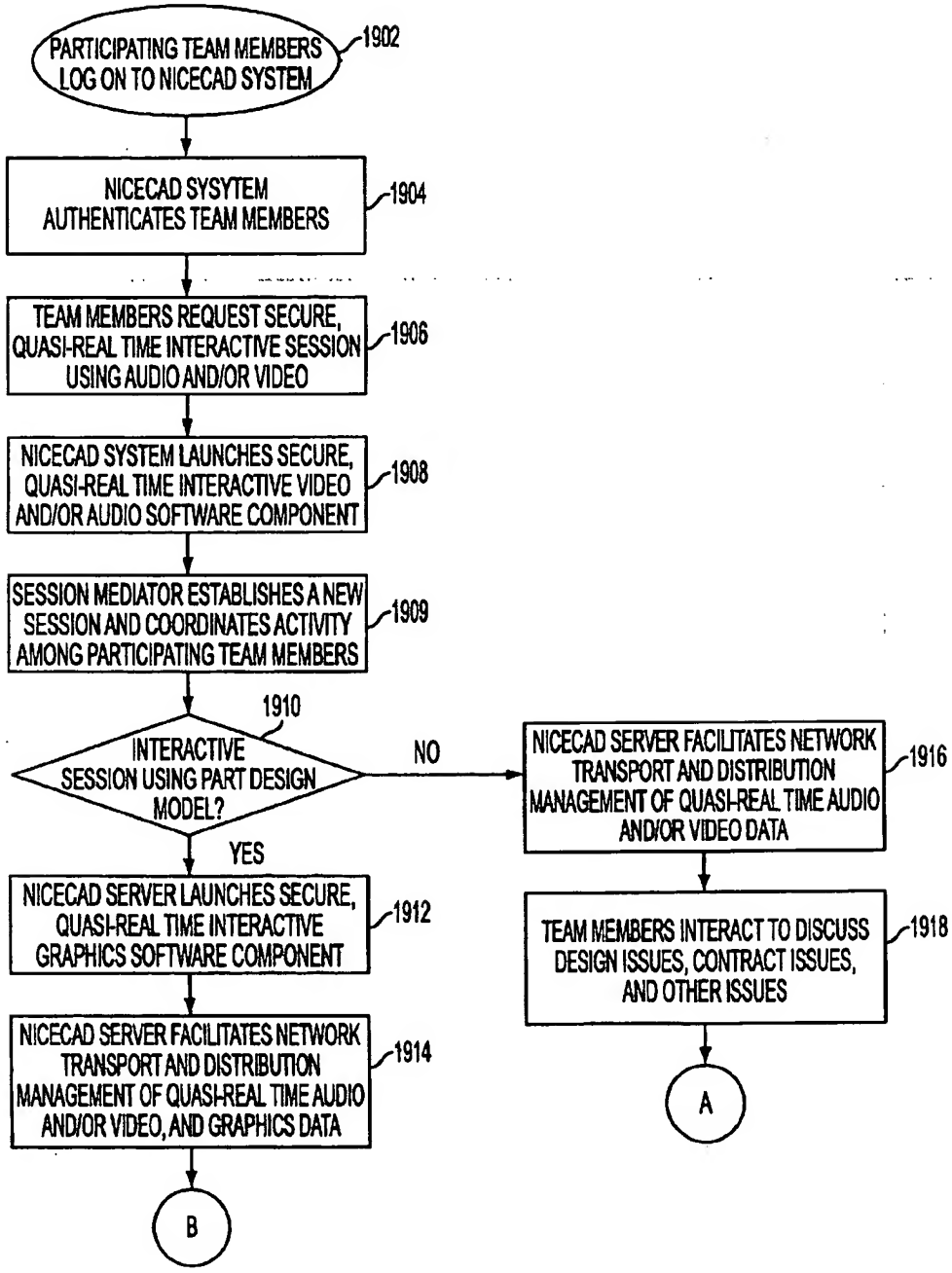

PARTICIPATING TEAM MEMBERS LOG ON TO NICECAD SYSTEM
1902
NICECAD SYSYTEM AUTHENTICATES TEAM MEMBERS
1904
TEAM MEMBERS REQUEST SECURE, QUASI-REAL TIME INTERACTIVE SESSION USING AUDIO AND/OR VIDEO
1906
NICECAD SYSTEM LAUNCHES SECURE, QUASI-REAL TIME INTERACTIVE VIDEO AND/OR AUDIO SOFTWARE COMPONENT
1908
SESSION MEDIATOR ESTABLISHES A NEW SESSION AND COORDINATES ACTIVITY AMONG PARTICIPATING TEAM MEMBERS
1909
1910
INTERACTIVE SESSION USING PART DESIGN MODEL?
NO
YES
NICECAD SERVER FACILITATES NETWORK TRANSPORT AND DISTRIBUTION MANAGEMENT OF QUASI-REAL TIME AUDIO AND/OR VIDEO DATA
1916
TEAM MEMBERS INTERACT TO DISCUSS DESIGN ISSUES, CONTRACT ISSUES, AND OTHER ISSUES
1918
A
NICECAD SERVER LAUNCHES SECURE, QUASI-REAL TIME INTERACTIVE GRAPHICS SOFTWARE COMPONENT
1912
NICECAD SERVER FACILITATES NETWORK TRANSPORT AND DISTRIBUTION MANAGEMENT OF QUASI-REAL TIME AUDIO AND/OR VIDEO, AND GRAPHICS DATA
1914
B

FIG. 19A

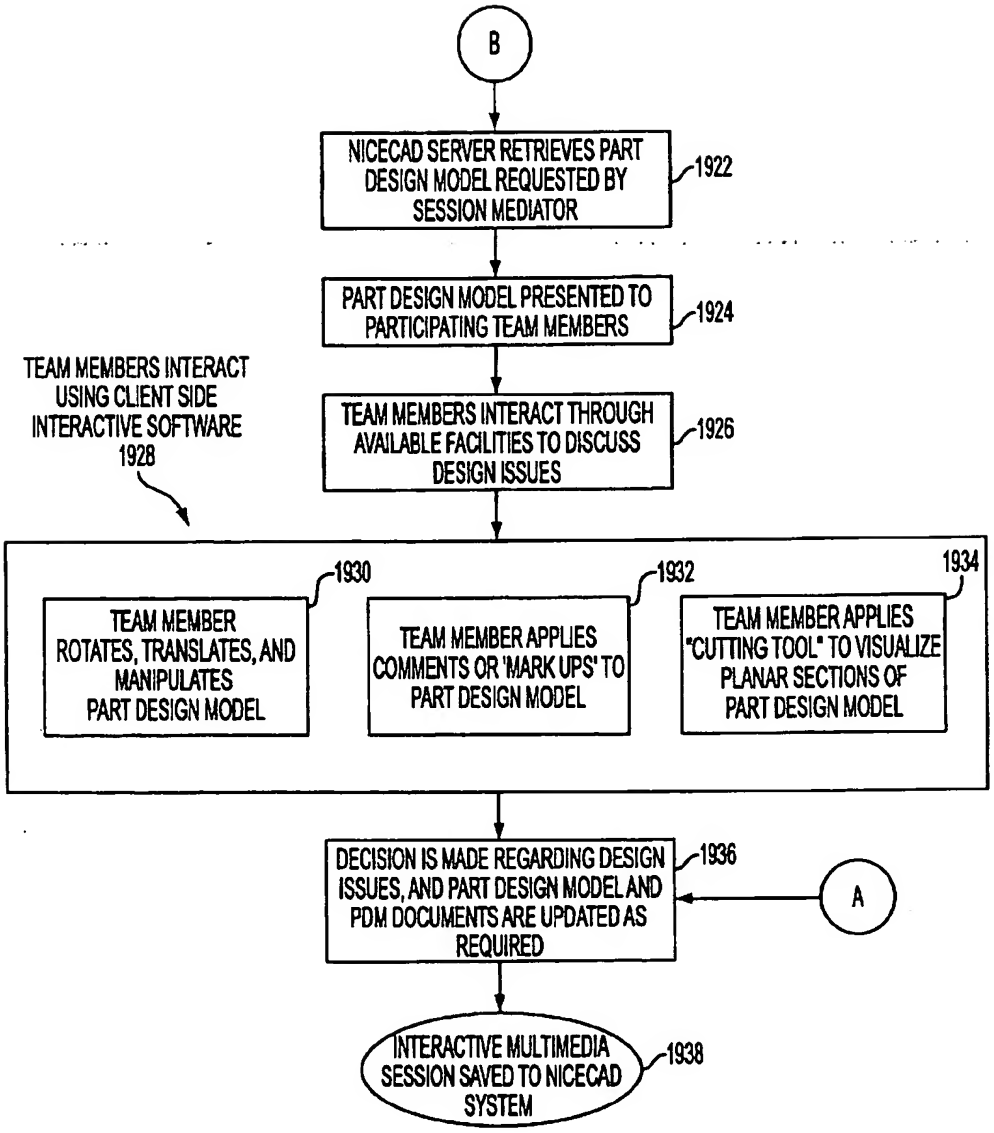

B
NICECAD SERVER RETRIEVES PART DESIGN MODEL REQUESTED BY SESSION MEDIATOR
1922
PART DESIGN MODEL PRESENTED TO PARTICIPATING TEAM MEMBERS
1924
TEAM MEMBERS INTERACT USING CLIENT SIDE INTERACTIVE SOFTWARE 1928
TEAM MEMBERS INTERACT THROUGH AVAILABLE FACILITIES TO DISCUSS DESIGN ISSUES
1926
1930
TEAM MEMBER ROTATES, TRANSLATES, AND MANIPULATES PART DESIGN MODEL
1932
TEAM MEMBER APPLIES COMMENTS OR 'MARK UPS' TO PART DESIGN MODEL
1934
TEAM MEMBER APPLIES "CUTTING TOOL" TO VISUALIZE PLANAR SECTIONS OF PART DESIGN MODEL
DECISION IS MADE REGARDING DESIGN ISSUES, AND PART DESIGN MODEL AND PDM DOCUMENTS ARE UPDATED AS REQUIRED
1936
A
INTERACTIVE MULTIMEDIA SESSION SAVED TO NICECAD SYSTEM
1938

FIG. 19B

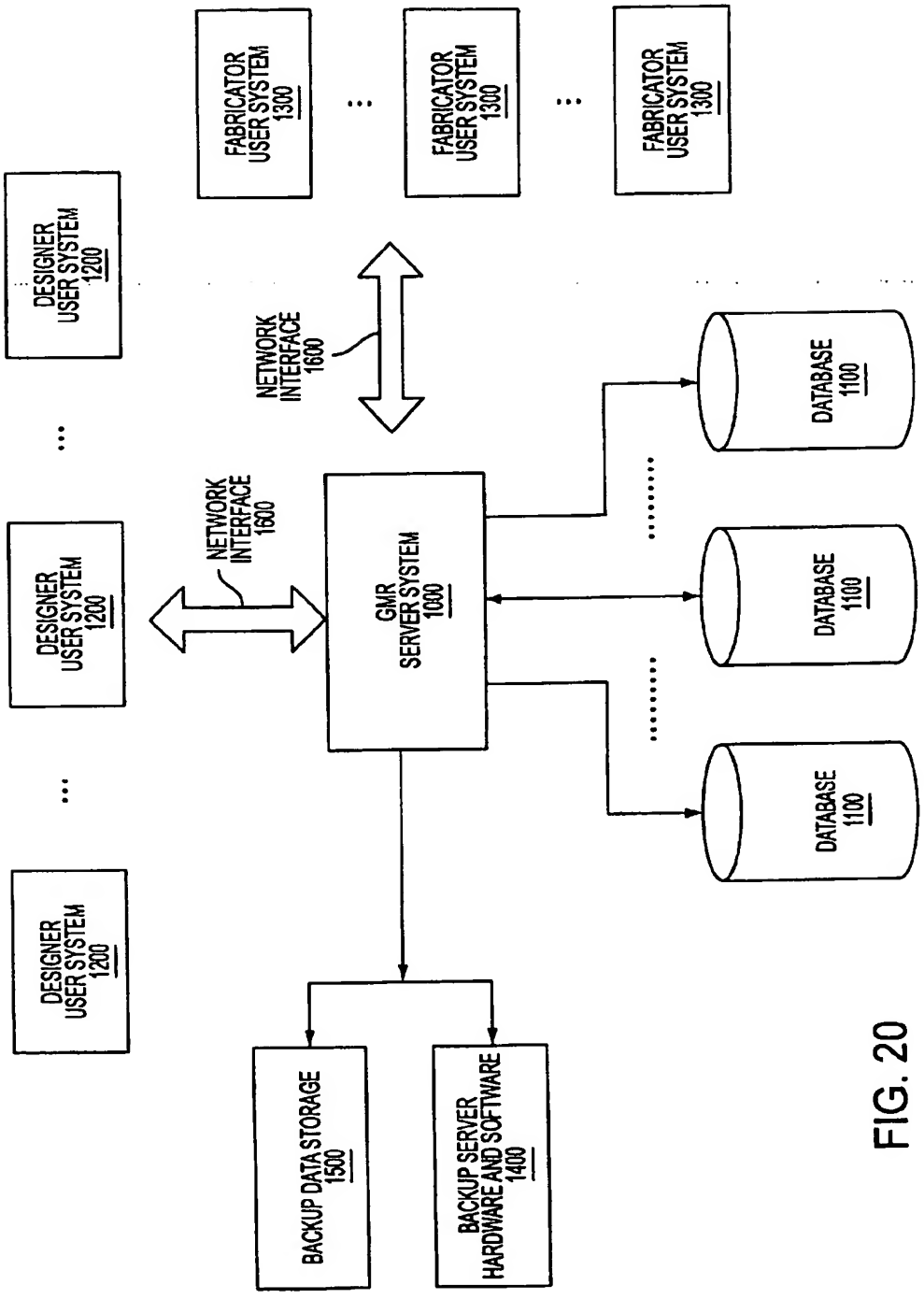

FABRICATOR USER SYSTEM 1300
FABRICATOR USER SYSTEM 1300
FABRICATOR USER SYSTEM 1300
DESIGNER USER SYSTEM 1200
DESIGNER USER SYSTEM 1200
DESIGNER USER SYSTEM 1200
NETWORK INTERFACE 1600
NETWORK INTERFACE 1600
GMR SERVER SYSTEM 1000
DATABASE 1100
DATABASE 1100
DATABASE 1100
BACKUP DATA STORAGE 1500
BACKUP SERVER HARDWARE AND SOFTWARE 1400

FIG. 20

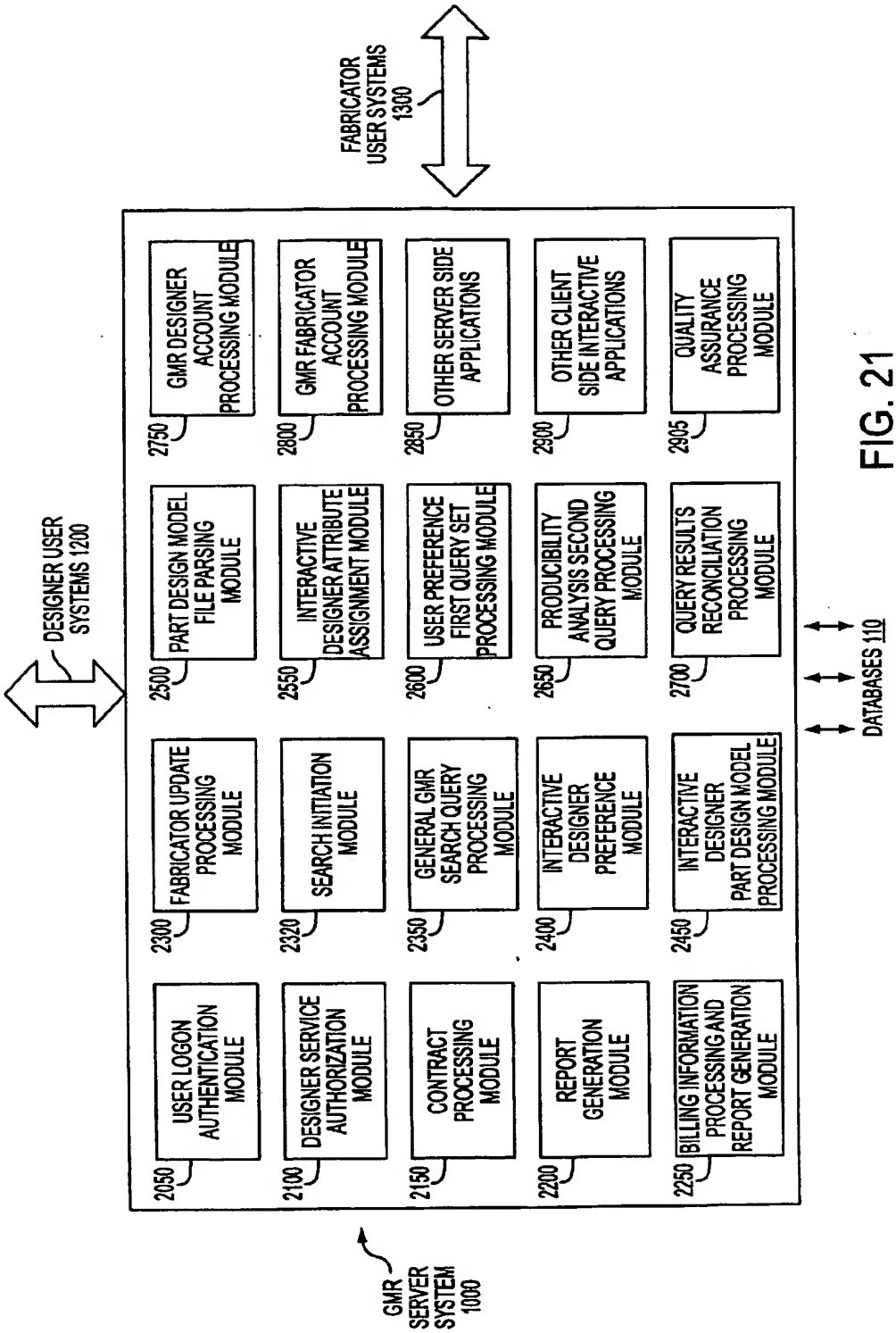

FABRICATOR USER SYSTEMS 1300
DESIGNER USER SYSTEMS 1200
GMR SERVER SYSTEM 1000
DATABASES 110
2750 GMR DESIGNER ACCOUNT PROCESSING MODULE
2800 GMR FABRICATOR ACCOUNT PROCESSING MODULE
2850 OTHER SERVER SIDE APPLICATIONS
2900 OTHER CLIENT SIDE INTERACTIVE APPLICATIONS
2905 QUALITY ASSURANCE PROCESSING MODULE
2500 PART DESIGN MODEL FILE PARSING MODULE
2550 INTERACTIVE DESIGNER ATTRIBUTE ASSIGNMENT MODULE
2600 USER PREFERENCE FIRST QUERY SET PROCESSING MODULE
2650 PRODUCIBILITY ANALYSIS SECOND QUERY PROCESSING MODULE
2700 QUERY RESULTS RECONCILIATION PROCESSING MODULE
2300 FABRICATOR UPDATE PROCESSING MODULE
2320 SEARCH INITIATION MODULE
2350 GENERAL GMR SEARCH QUERY PROCESSING MODULE
2400 INTERACTIVE DESIGNER PREFERENCE MODULE
2450 INTERACTIVE DESIGNER PART DESIGN MODEL PROCESSING MODULE
2050 USER LOGON AUTHENTICATION MODULE
2100 DESIGNER SERVICE AUTHORIZATION MODULE
2150 CONTRACT PROCESSING MODULE
2200 REPORT GENERATION MODULE
2250 BILLING INFORMATION PROCESSING AND REPORT GENERATION MODULE

FIG. 21

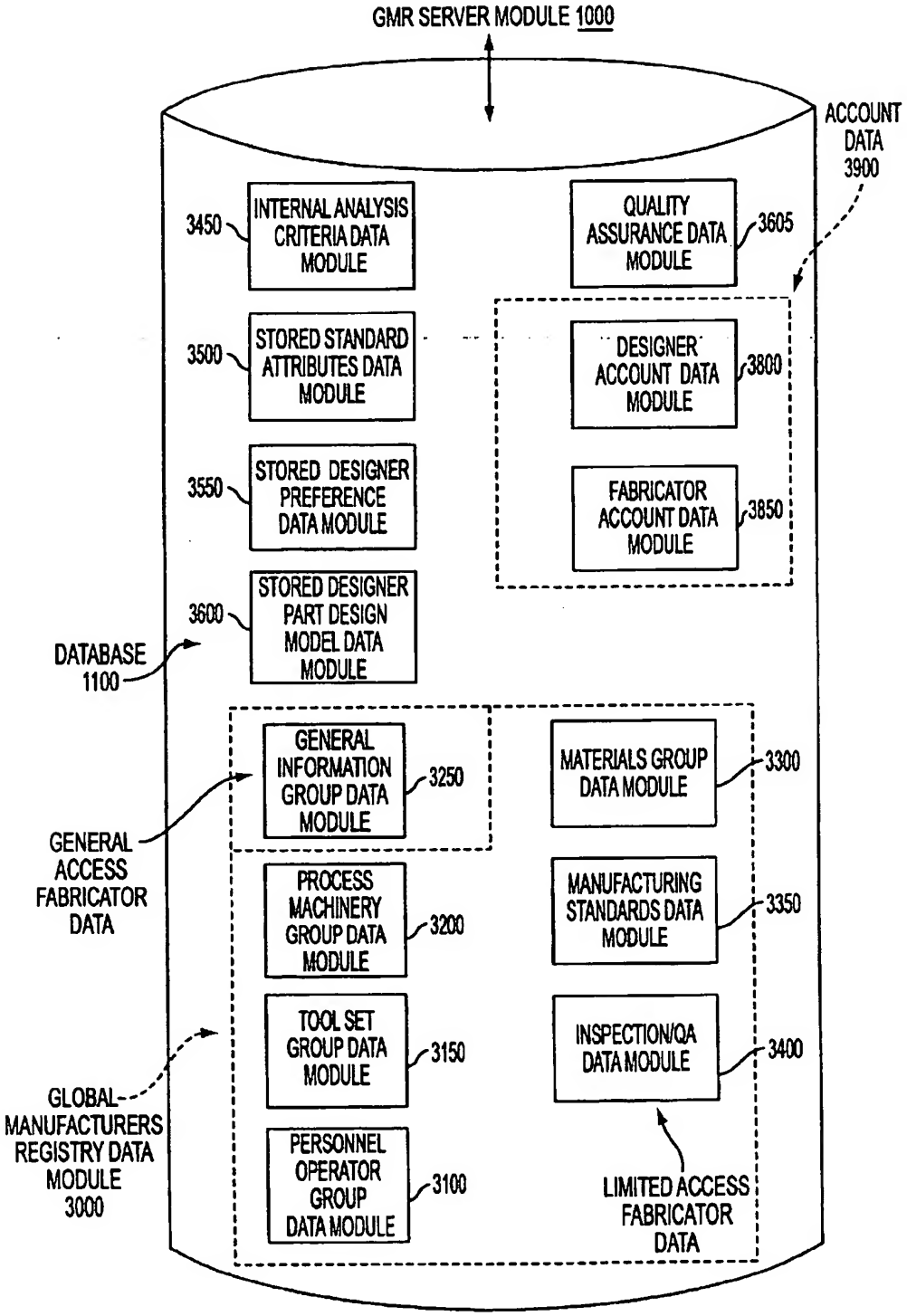

GMR SERVER MODULE 1000
ACCOUNT DATA 3900
3450
INTERNAL ANALYSIS CRITERIA DATA MODULE
QUALITY ASSURANCE DATA MODULE
3605
3500
STORED STANDARD ATTRIBUTES DATA MODULE
DESIGNER ACCOUNT DATA MODULE
3800
3550
STORED DESIGNER PREFERENCE DATA MODULE
FABRICATOR ACCOUNT DATA MODULE
3850
3600
STORED DESIGNER PART DESIGN MODEL DATA MODULE
DATABASE 1100
GENERAL INFORMATION GROUP DATA MODULE
3250
MATERIALS GROUP DATA MODULE
3300
GENERAL ACCESS FABRICATOR DATA
PROCESS MACHINERY GROUP DATA MODULE
3200
MANUFACTURING STANDARDS DATA MODULE
3350
TOOL SET GROUP DATA MODULE
3150
INSPECTION/QA DATA MODULE
3400
GLOBAL MANUFACTURERS REGISTRY DATA MODULE 3000
PERSONNEL OPERATOR GROUP DATA MODULE
3100
LIMITED ACCESS FABRICATOR DATA

FIG. 22

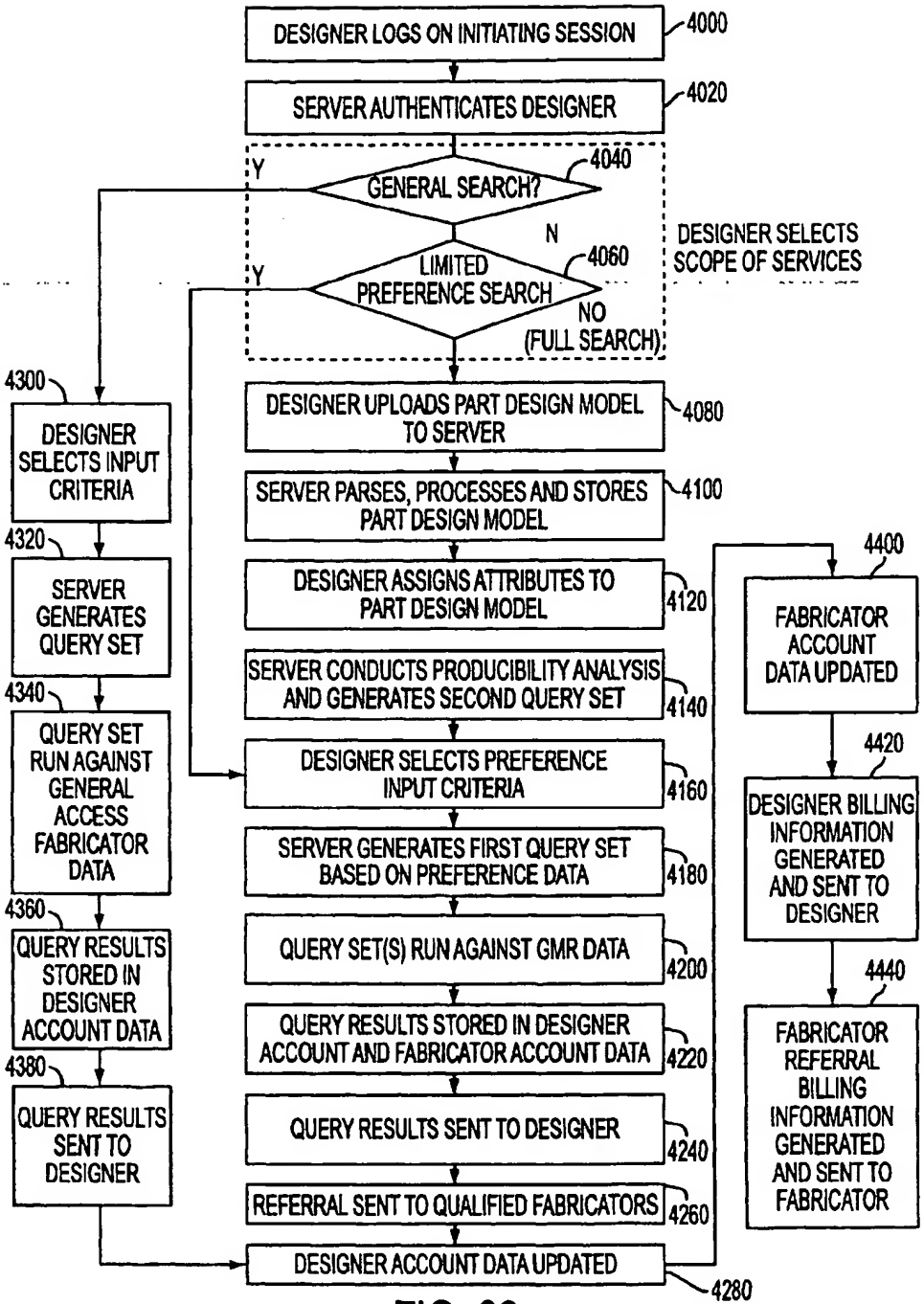

DESIGNER LOGS ON INITIATING SESSION
4000
SERVER AUTHENTICATES DESIGNER
4020
GENERAL SEARCH?
4040
Y
N
LIMITED PREFERENCE SEARCH
4060
Y
NO (FULL SEARCH)
DESIGNER SELECTS SCOPE OF SERVICES
4300
DESIGNER SELECTS INPUT CRITERIA
4320
SERVER GENERATES QUERY SET
4340
QUERY SET RUN AGAINST GENERAL ACCESS FABRICATOR DATA
4360
QUERY RESULTS STORED IN DESIGNER ACCOUNT DATA
4380
QUERY RESULTS SENT TO DESIGNER
DESIGNER UPLOADS PART DESIGN MODEL TO SERVER
4080
SERVER PARSES, PROCESSES AND STORES PART DESIGN MODEL
4100
DESIGNER ASSIGNS ATTRIBUTES TO PART DESIGN MODEL
4120
SERVER CONDUCTS PRODUCIBILITY ANALYSIS AND GENERATES SECOND QUERY SET
4140
DESIGNER SELECTS PREFERENCE INPUT CRITERIA
4160
SERVER GENERATES FIRST QUERY SET BASED ON PREFERENCE DATA
4180
QUERY SET(S) RUN AGAINST GMR DATA
4200
QUERY RESULTS STORED IN DESIGNER ACCOUNT AND FABRICATOR ACCOUNT DATA
4220
QUERY RESULTS SENT TO DESIGNER
4240
REFERRAL SENT TO QUALIFIED FABRICATORS
4260
DESIGNER ACCOUNT DATA UPDATED
4280
4400
FABRICATOR ACCOUNT DATA UPDATED
4420
DESIGNER BILLING INFORMATION GENERATED AND SENT TO DESIGNER
4440
FABRICATOR REFERRAL BILLING INFORMATION GENERATED AND SENT TO FABRICATOR

FIG. 23

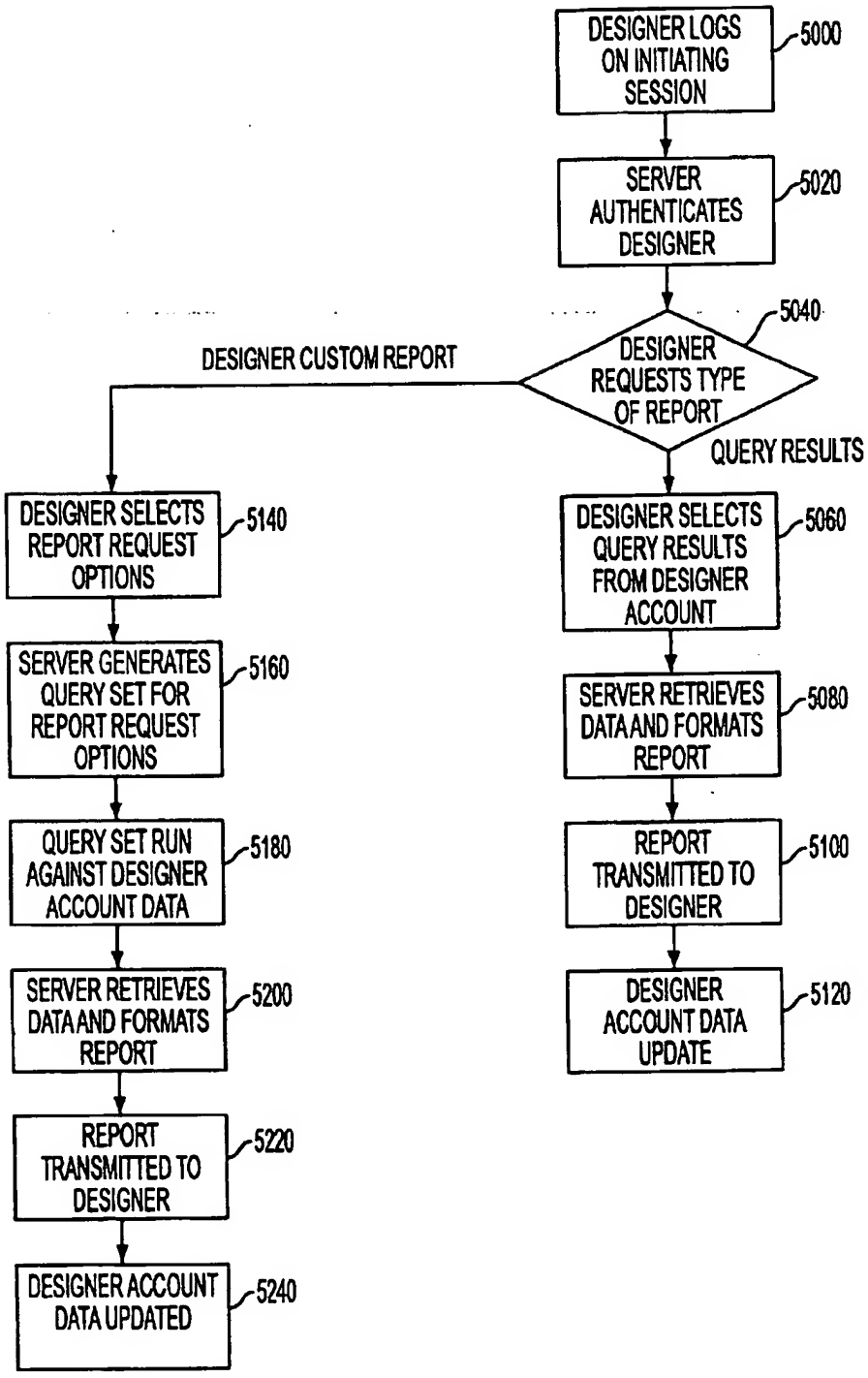

DESIGNER LOGS ON INITIATING SESSION
5000
SERVER AUTHENTICATES DESIGNER
5020
5040
DESIGNER REQUESTS TYPE OF REPORT
DESIGNER CUSTOM REPORT
QUERY RESULTS
DESIGNER SELECTS REPORT REQUEST OPTIONS
5140
SERVER GENERATES QUERY SET FOR REPORT REQUEST OPTIONS
5160
QUERY SET RUN AGAINST DESIGNER ACCOUNT DATA
5180
SERVER RETRIEVES DATA AND FORMATS REPORT
5200
REPORT TRANSMITTED TO DESIGNER
5220
DESIGNER ACCOUNT DATA UPDATED
5240
DESIGNER SELECTS QUERY RESULTS FROM DESIGNER ACCOUNT
5060
SERVER RETRIEVES DATA AND FORMATS REPORT
5080
REPORT TRANSMITTED TO DESIGNER
5100
DESIGNER ACCOUNT DATA UPDATE
5120

FIG. 24

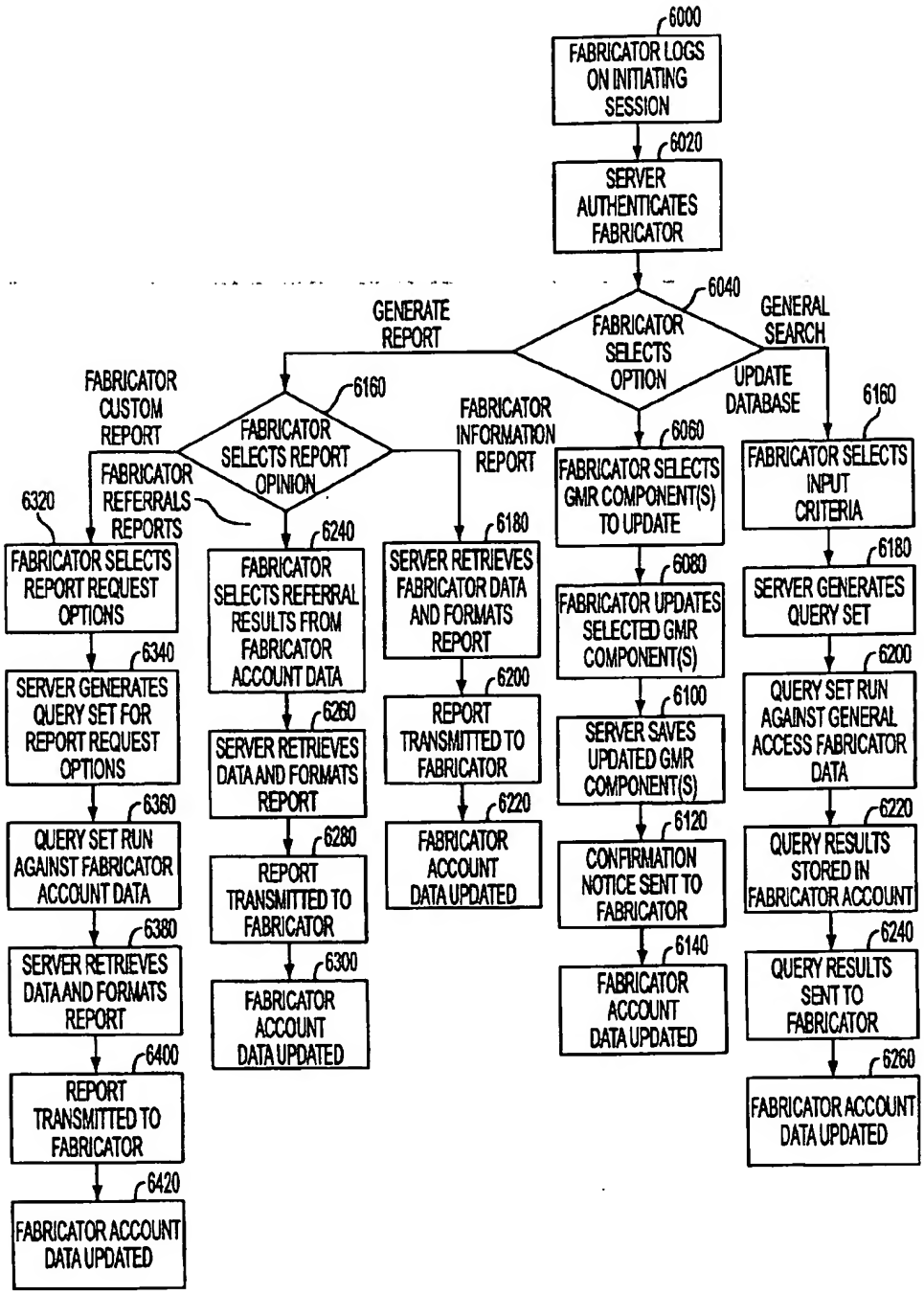

6000
FABRICATOR LOGS ON INITIATING SESSION
6020
SERVER AUTHENTICATES FABRICATOR
6040
FABRICATOR SELECTS OPTION
GENERATE REPORT
GENERAL SEARCH
UPDATE DATABASE
FABRICATOR CUSTOM REPORT
6160
FABRICATOR SELECTS REPORT OPINION
FABRICATOR REFERRALS REPORTS
FABRICATOR INFORMATION REPORT
6320
FABRICATOR SELECTS REPORT REQUEST OPTIONS
6340
SERVER GENERATES QUERY SET FOR REPORT REQUEST OPTIONS
6360
QUERY SET RUN AGAINST FABRICATOR ACCOUNT DATA
6380
SERVER RETRIEVES DATA AND FORMATS REPORT
6400
REPORT TRANSMITTED TO FABRICATOR
6420
FABRICATOR ACCOUNT DATA UPDATED
6240
FABRICATOR SELECTS REFERRAL RESULTS FROM FABRICATOR ACCOUNT DATA
6260
SERVER RETRIEVES DATA AND FORMATS REPORT
6280
REPORT TRANSMITTED TO FABRICATOR
6300
FABRICATOR ACCOUNT DATA UPDATED
6180
SERVER RETRIEVES FABRICATOR DATA AND FORMATS REPORT
6200
REPORT TRANSMITTED TO FABRICATOR
6220
FABRICATOR ACCOUNT DATA UPDATED
6060
FABRICATOR SELECTS GMR COMPONENT(S) TO UPDATE
6080
FABRICATOR UPDATES SELECTED GMR COMPONENT(S)
6100
SERVER SAVES UPDATED GMR COMPONENT(S)
6120
CONFIRMATION NOTICE SENT TO FABRICATOR
6140
FABRICATOR ACCOUNT DATA UPDATED
6160
FABRICATOR SELECTS INPUT CRITERIA
6180
SERVER GENERATES QUERY SET
6200
QUERY SET RUN AGAINST GENERAL ACCESS FABRICATOR DATA
6220
QUERY RESULTS STORED IN FABRICATOR ACCOUNT
6240
QUERY RESULTS SENT TO FABRICATOR
6260
FABRICATOR ACCOUNT DATA UPDATED

FIG. 25

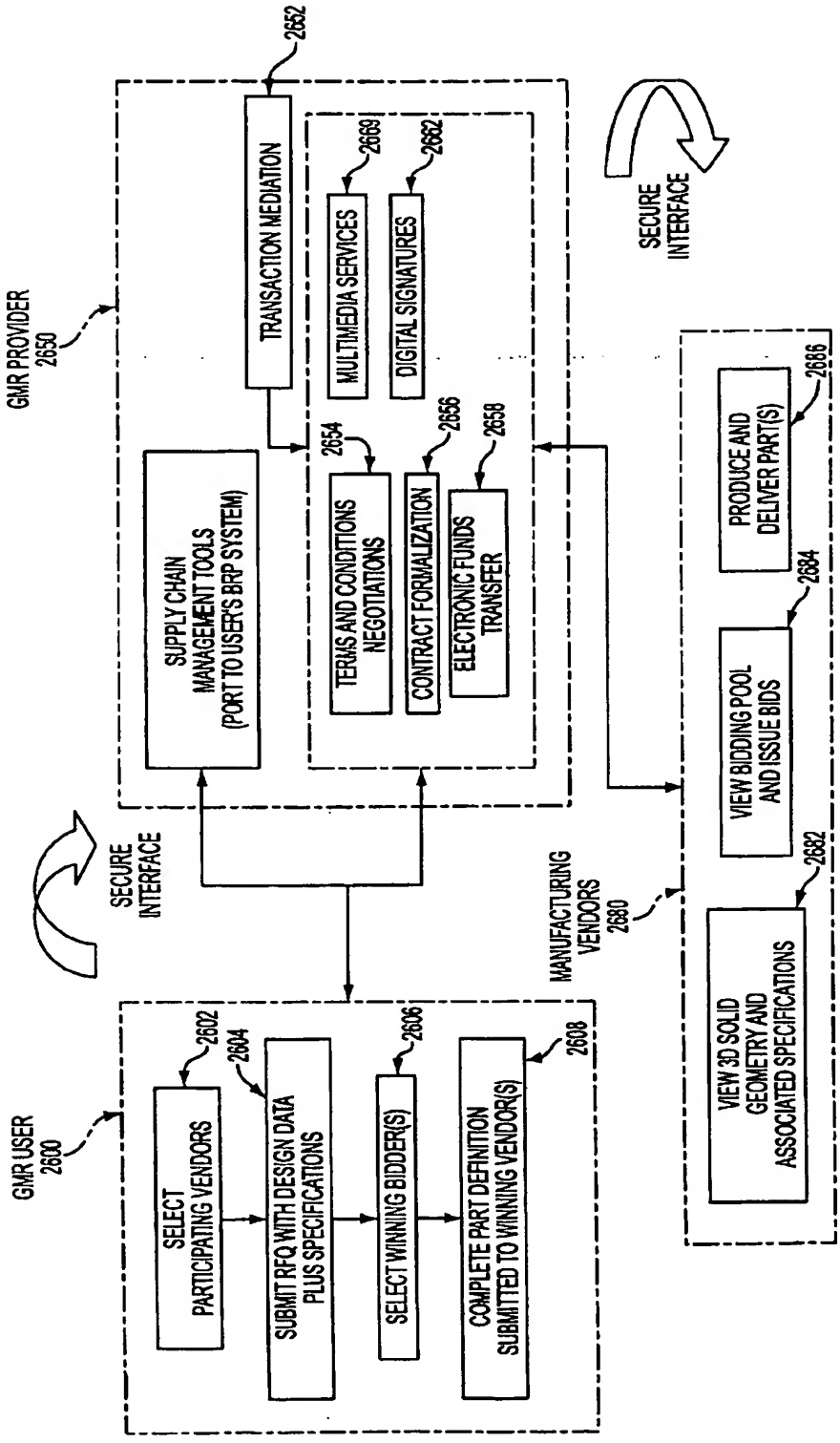

GMR USER
2600
SELECT PARTICIPATING VENDORS
2602
SUBMIT RFQ WITH DESIGN DATA PLUS SPECIFICATIONS
2604
SELECT WINNING BIDDER(S)
2606
COMPLETE PART DEFINITION SUBMITTED TO WINNING VENDOR(S)
2608
SECURE INTERFACE
GMR PROVIDER
2650
SUPPLY CHAIN MANAGEMENT TOOLS (PORT TO USER'S BRP SYSTEM)
TRANSACTION MEDIATION
2652
TERMS AND CONDITIONS NEGOTIATIONS
2654
CONTRACT FORMALIZATION
2656
ELECTRONIC FUNDS TRANSFER
2658
MULTIMEDIA SERVICES
2660
DIGITAL SIGNATURES
2662
SECURE INTERFACE
MANUFACTURING VENDORS
2680
VIEW 3D SOLID GEOMETRY AND ASSOCIATED SPECIFICATIONS
2682
VIEW BIDDING POOL AND ISSUE BIDS
2684
PRODUCE AND DELIVER PART(S)
2686

FIG. 26

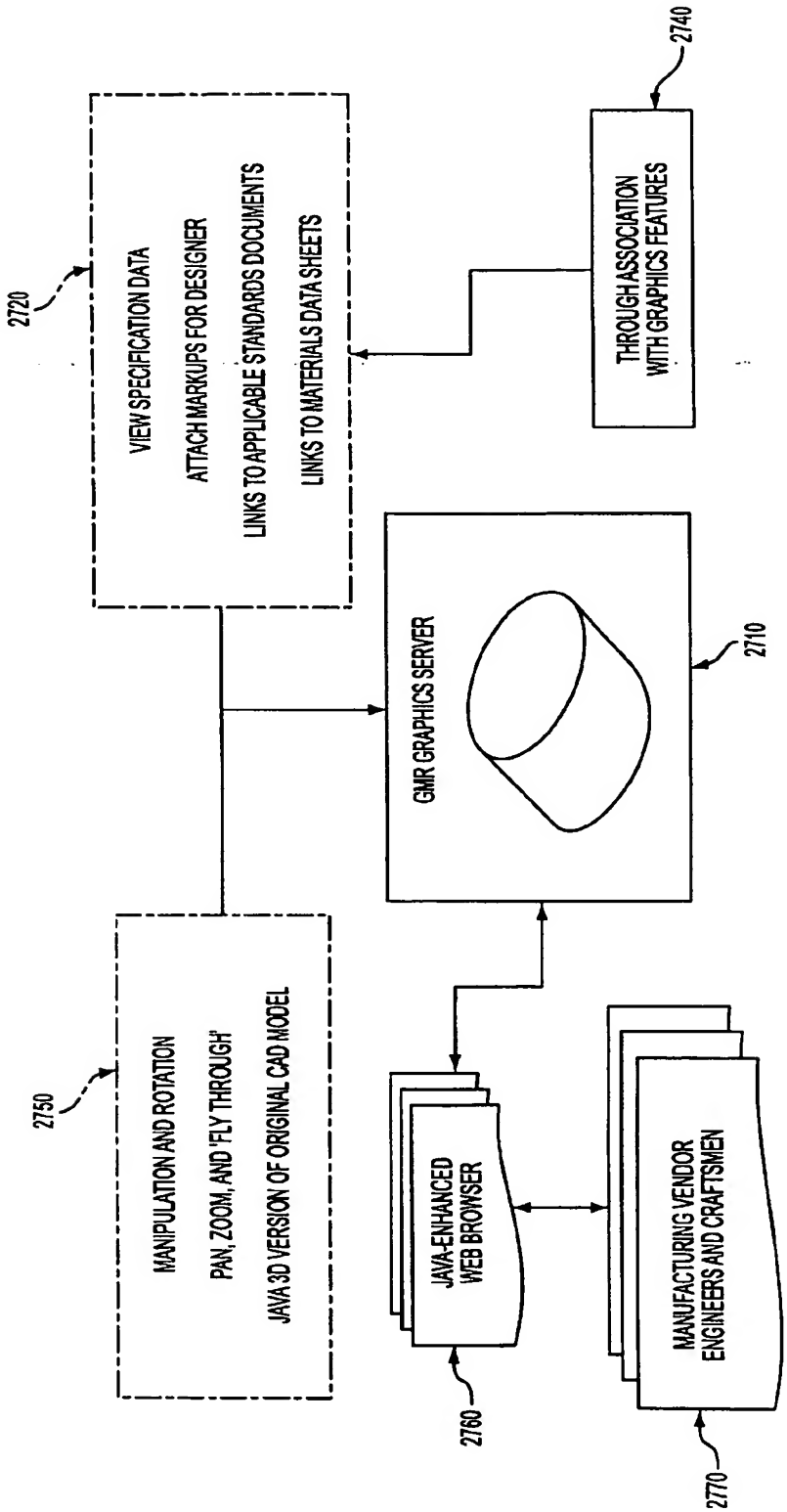

2740
THROUGH ASSOCIATION WITH GRAPHICS FEATURES
2720
VIEW SPECIFICATION DATA
ATTACH MARKUPS FOR DESIGNER
LINKS TO APPLICABLE STANDARDS DOCUMENTS
LINKS TO MATERIALS DATA SHEETS
GMR GRAPHICS SERVER
2710
2750
MANIPULATION AND ROTATION
PAN, ZOOM, AND 'FLY THROUGH'
JAVA 3D VERSION OF ORIGINAL CAD MODEL
JAVA-ENHANCED WEB BROWSER
2760
MANUFACTURING VENDOR ENGINEERS AND CRAFTSMEN
2770

FIG. 27

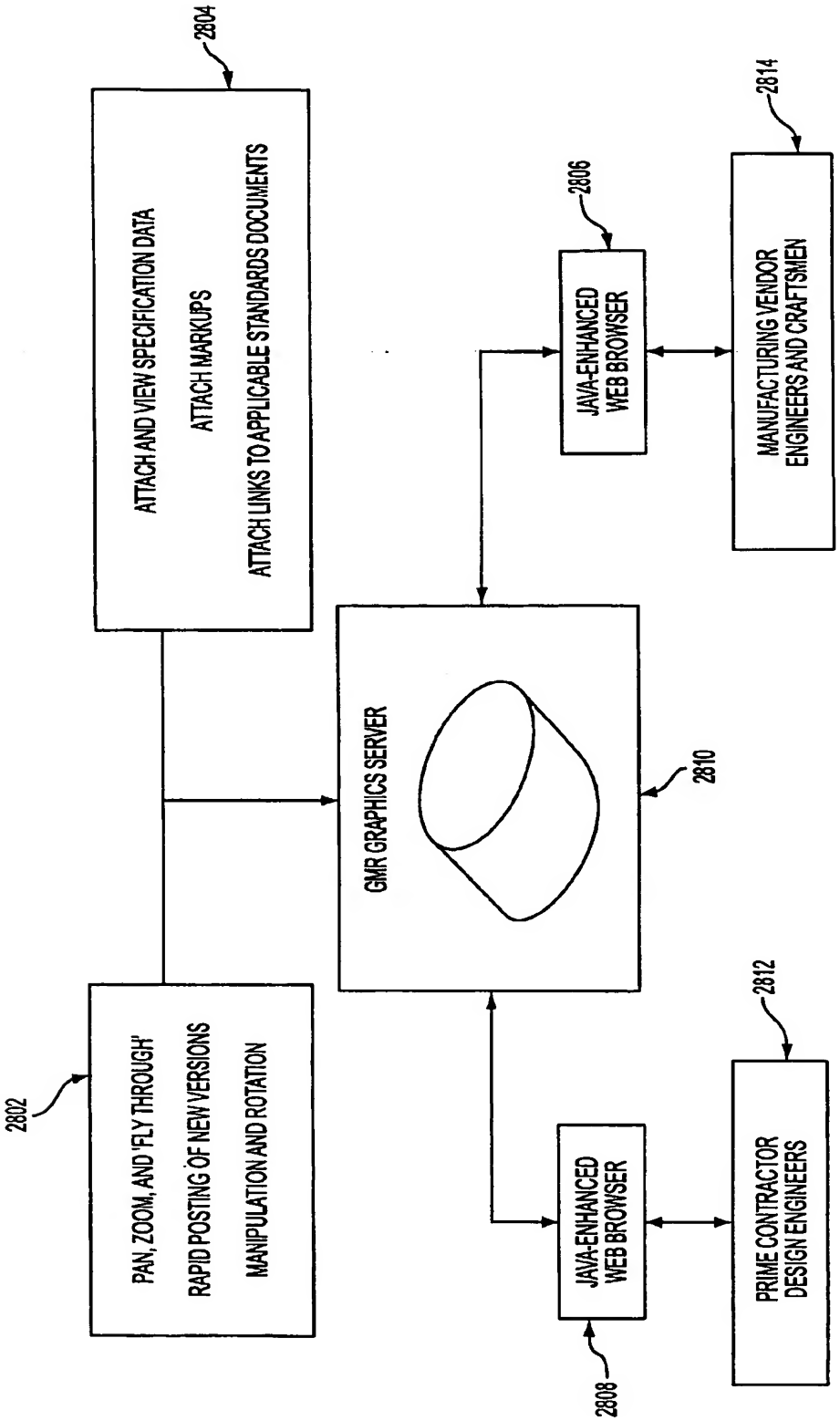

2804
ATTACH AND VIEW SPECIFICATION DATA
ATTACH MARKUPS
ATTACH LINKS TO APPLICABLE STANDARDS DOCUMENTS
2806
JAVA-ENHANCED WEB BROWSER
2814
MANUFACTURING VENDOR ENGINEERS AND CRAFTSMEN
GMR GRAPHICS SERVER
2810
2802
PAN, ZOOM, AND 'FLY THROUGH'
RAPID POSTING OF NEW VERSIONS
MANIPULATION AND ROTATION
2812
JAVA-ENHANCED WEB BROWSER
PRIME CONTRACTOR DESIGN ENGINEERS
2808

FIG. 28

NETWORK-BASED SYSTEM FOR THE MANUFACTURE OF PARTS WITH A VIRTUAL COLLABORATIVE ENVIRONMENT FOR DESIGN, DEVELOPEMENT, AND FABRICATOR SELECTION

RELATED APPLICATIONS

This application is a continuation-in-part of U.S. application Ser. No. 09/270,007, filed Mar. 16, 1999, entitled "Interactive System for Engineering Design and Manufacture," and U.S. application Ser. No. 09/311,150, filed May 13, 1999, entitled "Network Integrated Concurrent Engineering With Computer Aided Design," both of which are herein incorporated by reference.

FIELD OF THE INVENTION

The present invention relates generally to a comprehensive, integrated computer-based system and method for undertaking an engineering design and development effort in a virtual collaborative environment, identifying qualified fabricators for manufacturing a part design based on fabricator capability information stored in a global registry database substantially maintained by the fabricators themselves, and conducting a virtual bidding process whereby electronic representations of three dimensional model and specification data are provided by a central server. The central server further supports the bidding process by providing quasi-real time audio, video and graphics, and the contracts negotiation and formalization steps.

BACKGROUND OF THE INVENTION

The conventional approach to an engineering design and development effort tends to be costly and cumbersome. Typically, a design and development effort begins with an initial idea or concept for a new product, for example, an improved sonobuoy for receiving and transmitting to a surveiling aircraft acoustic data from the ocean.

From this initial idea, an initial design comprising drawings and written specifications is created. The drawings may be paper drawings or "blueprints"or may be three dimensional drawings created using Computer Aided Design (CAD) software. The initial design will usually include certain specifications, such as product specifications (defining performance of the item), system specifications (defining performance of a system including the item) and interface design specifications (defining electronic or physical interfaces between the item and other system components). In the case of a sonobuoy, these documents will include a product specification detailing the performance criteria and "form, fit and function" of the sonobuoy; a system specification detailing the performance of the system including the sonobuoy and aircraft avionics; and an interface design specification defining the electronic interface between the sonobuoy and the aircraft receiver and specialized acoustical processors.

These specifications will include various design criteria or performance parameters for the new sonobuoy, such as reliability requirements, mean time between failure, radio frequency (RF) transmission power and range, acoustic gain and performance (e.g., gain in decibels and three dimensional beam patterns) and so on.

As depicted in FIG. 1, this initial idea or concept is translated into an initial design, "Design 1." A series of teams, comprising one or more members, are assembled to

evaluate the design according to their various engineering, business and management specialties. Typically, there will be a management team whose primary function is to ensure that the development program is "on schedule" and "on cost." There will be a business/accounting team who will analyze the costs of the development program and the expected costs of the product produced in quantity. There will be a series of engineering teams, each of a particular discipline or specialty. In our sonobuoy example, there will be mechanical engineers, electrical engineers, RF engineers, acoustic specialists, reliability engineers, safety engineers, signal processing specialists, production engineers and so on.

Each of these teams will evaluate the design (three dimensional model and associated specifications) to determine conformance with requirements. In a typical development effort, these teams will recommend changes to the design to rectify deficiencies or improve performance. The design will evolve (as depicted in FIG. 1, from Design 1, Design 2, and so on to final Design n) as changes are recommended and implemented.

Because a number of disciplines may be involved, it can be appreciated that the process can be lengthy and costly. When one team implements changes based on its analysis, other teams may have to redo their analysis as a result. For example, when a production engineer recommends (based on cost or manufacturibility considerations) that paper capacitors be employed instead of ceramic capacitors, the reliability engineer will need to reevaluate the design. When an acoustic specialist recommends that improved hydrophones be used or an RF specialist recommends that an improved receiver be used, the mechanical engineer will need to determine if packaging limits are exceeded and the accounting specialists will need to determine if "design to cost" parameters are exceeded.

Application of the "concurrent engineering" principle can improve the process somewhat by assembling the various specialists at the earliest possible time. Thus, manufacturing specialists are consulted from the beginning, rather than simply at the end.

Nevertheless, the sonobuoy example and FIG. 1 illustrate how a complex engineering effort involving multiple disciplines can be a lengthy and costly process as designs are considered and discarded, new designs are evaluated and so forth. When the various specialists are in different geographical locations ("physical boundaries"), at different business entities ("business boundaries"), or employ different software standards ("format boundaries"), the problem is only compounded. The disparate locations, business cultures and format standards can be significant impediments. This can be referred to as the "boundaries problem." Put simply, having specialists of differing disciplines (with often conflicting priorities) attempt to resolve design issues when operating from different locations, or from business entities with differing business practices and cultures, or when using different software formats (e.g., three dimensional model standards) creates significant costs and obstacles. This is a significant drawback.

Once an acceptable design is arrived at (e.g., Design n, FIG. 1), the developing concern typically wishes to produce the design in quantity. In today's decentralized economy, where most "start-up" or even moderately-sized engineering enterprises do not have their own production facilities, this typically requires "outsourcing" of the production. In some respects, the process of locating qualified, performance-proven fabricators can be as daunting as the engineering

development effort. The engineering concern may not have established relationships with many—or any—manufacturers. Of the few potential candidates that may be identified through "word of mouth" or a costly search, it can be costly to evaluate whether such candidates meet minimum requirements. Numerous meetings may be required. Even then, it may be difficult to ascertain the quality of past performance and the prospects for the proposed performance. The engineering concern will have to sustain the cost, and risk, of divulging proprietary design and specification data. Because format standards may differ, the costs and risks of (sometimes imperfect) file conversions may be required. These are significant drawbacks.

Even if several acceptable fabricators are located, they may represent but a fraction of the otherwise qualified pool of fabricators. This lessens competition and ultimately, can lead to increased costs and decreased performance. These are significant disadvantages to the engineering concern. With respect to start-up or not-well-known fabricators, this is a significant drawback that prevents them from penetrating new markets and inhibits their growth.

Once a pool (or even a single) of qualified fabricators is identified, the process of negotiating an agreement on performance must occur. This may require numerous phonecalls, teleconference calls, and face-to-face meetings. Three dimensional models and specification documents may have to be divulged so that the fabricator can develop a bid. Despite rigorous concurrent engineering, it is common that minor and not-so-minor redesigns may be required due to producibility concerns. This will require an additional engineering effort between the supplier and the proposed fabricator to discuss and decide on engineering changes to arrive at a mutually acceptable, and producible, design that meets and performance and cost requirements. The time and effort entailed are significant drawbacks.

Engineering issues aside, the negotiation process can be lengthy and costly. Documents, which may be proprietary, are sent to the bidding fabricator. The bidding fabricator must evaluate them, engage in discussions with the prime contractor about the design and specifications, and generate a "bid." This bid typically includes terms relating to time, cost and performance. Upon receipt of the bid, additional discussions between the parties may be required to address engineering and/or contractual issues. Typically conducted using a combination of the phone, teleconference and postal service (or the like), these discussions impose additional costs and time in getting a product to market. This is a significant drawback.

Other problems and drawbacks also exist.

SUMMARY OF THE INVENTION

For these and like reasons, what is desired is a network-based, interactive system capable of receiving specification and other information so that an engineering design for a product, represented by an electronic three-dimensional model, can be designed, developed and evaluated in a collaborative, virtual environment substantially obviating geographic, business and format boundaries.

Accordingly, it is one object of the present invention to overcome one or more of the aforementioned and other limitations of existing systems and methods for undertaking an engineering development effort.

It is another object of the present invention to provide such a network-based system whereby a central server maintains engineering data, such as design documents and three dimensional model data, in a common, neutral format, which is accessible by authorized team members through a graphical user interface that is substantially platform independent to reduce or eliminate the necessity for specialized hardware and software.

It is another object of the present invention for a central server to support a secure multimedia communications capability to include audio, video and graphics, so that participants in an engineering effort can communicate and collaborate in the virtual engineering environment to discuss a design.

It is another object of the present invention to provide a central server with a repository of fabricator capability data which can be searched by a prime contractor to locate qualified fabricators based on general criteria as well as design-specific criteria.

It is another object of the present invention to provide a central server whereby said fabricator capability data is substantially maintained by the fabricators so as to incentivize them to participate in the so-called manufacturer's registry because it provides them a capability management system.

It is another object of the present invention to provide such a searchable manufacturer's registry further supporting the collection of quality assurance data based on fabricator performance, the quality assurance data being provided to the fabricator and to future prime contractors considering that fabricator.

It is another object of the present invention to provide an electronic marketplace and bidding system whereby a prime contractor can solicit proposals or bids in a virtual environment including the provision of specification and three-dimensional model data to prospective fabricators for consideration in developing their proposals.

It is another object of the present invention to provide such an electronic marketplace whereby a prime contractor and prospective fabricators can discuss design and contractual issues in a virtual environment using quasi- real-time audio, video and graphics, including the three-dimensional model representing the part or product.

To achieve these and other objects of the present invention, and in accordance with the purpose of the invention, as embodied and as broadly described, an embodiment of the present invention comprises an apparatus and method for a network-based interactive system that supports several phases of an engineering effort: the development and evaluation of an engineering design, the identification of potentially qualified fabricators, and the bidding and negotiation process to create an agreement for a qualified fabricator to manufacture a design in quantity.

Accordingly, what is disclosed is a network-based system interfacing multiple user systems which interface through a central server to undertake the design development effort. A baseline design is created and maintained in a neutral or common format by the central server. A governing entity or prime contractor assigns access or authorization data so that parts or the entirety of the three dimensional model and/or specification data is accessible by other team members in order to perform various analysis and simulations. An integrated product data management (PDM) capability manages access to controlled data and maintains a record of the various manifestations of the design. A current baseline design is maintained so that engineering analysis and simulation team members perform their analysis on the correct design. The system supports quasi- real-time interactive audio, video and graphics so that team members can discuss design issues in the virtual environment without having to

cross geographic or format boundaries. The above-described capabilities are provided in a substantially platform independent manner using a graphical user interface (GUI) supported by standard software, such as a "browser," which may further include machine independent applications, such as Java Applets™, or, for certain required tools, a semi-machine-dependent application, to reduce the processing burden on user stations and the need for specialized client-side software. In an alternative embodiment of the invention, a so-called "custom browser" may be provided to user stations so that the interface with the central server system is tailored to the present application. In yet another embodiment, certain graphics-intensive applications may be provided by the central server as client-side applications to be compiled and executed, such as applications coded in Open GL™ by Silicon Graphics™. The so-called common denominator in these embodiments is the goal of retaining the most calculation intensive tasks on the server side of the overall system so as to reduce the need for specialized software and hardware capabilities at the user systems. Additionally, by receiving, converting, and maintaining part design models in a common, neutral format, the central server system provides the important benefit of "data neutrality" for the participants in a concurrent engineering development project. In short, format boundaries are obviated. This is an important benefit, especially for small enterprises, which might otherwise be eliminated from participation in a project because their software resource do not support a format required by a prime contractor.

A second aspect of the system provides a database of fabricators, a so-called "Global Manufacturer's Registry," which can be searched by a designer or prime contractor to identify qualified fabricators. The database contains various information describing the fabricators, such as location, experience, machinery and process capabilities, certifications, quality assurance/inspection standards, and so forth. Fabricators are incentivized to enroll in the registry because it provides an incidental benefit as a capability management system they can use to manage and track their resources and capabilities. A prime contractor with a part design model representing a product can conduct a search to identify qualified fabricators based on design-specific and general criteria. The system includes producibility logic to analyze an uploaded part design in order to help identify qualified fabricators. The system allows prime contractors to provide feedback on performance which can be factored into a quality assurance capability provided by the system. As with the NICECAD aspect of the integrated system, the GMR aspect provides data neutrality for users by supporting the upload and conversion of part design models from various format types into a standard neutral format. Therefore, a designer is not precluded from using the GMR system based on the fact that it uses a particular part design model format.

According to a third aspect of the system, an electronic bidding system is provided by an Electronic Trading Community (ETC) to allow virtual discussions and negotiations to take place over the networked system once a pool of qualified fabricators is identified. A request for quote (RFQ) or request for proposal (RFP) with design data is submitted to the system so that fabricators can submit proposals. The virtual computer aided design (CAD) capability allows quasi- real-time discussions, including audio, video and graphics. The graphics capability allows a prime contractor and prospective fabricator to view the three-dimensional part design, including the execution of various manipulations, such as virtual rotations and translations, pan, zoom and "fly throughs." The stored part design model data may include links to associated specifications, standards and other design specific documents so that the bidding fabricator has a full representation of what it is bidding on. As with the NICECAD and GMR aspects of the integrated system, the ETC provides "data neutrality" so that format boundaries do not become impediments for either the soliciting prime contractor/designer or the bidding fabricator.

In general, the above capabilities are provided in a networked, virtual environment that reduces transaction costs and meets engineering challenges as it transcends geographic, business and format boundaries. One or more central servers maintain the part design model in neutral format, such as AP 214 STEP format, well known to those of skill in the art. Substantially platform user independent graphical interfaces, such as browser pages, and server-client interactive applications, such as Java Applets™ make the system substantially user-system independent. In some instances, instead of using Java™ mini-applications or the like to run on a browser's so-called "virtual machine," certain graphics-intensive or computation-intensive applications may be compiled and executed at the user systems as semi- machine-dependent applications amenable to operations using a computer system with a standard operating system. Whether specialized applications are supported by miniapplications (e.g., such as those coded in Java™) provided through a browser or by server interface with a semi-machine-dependent application at a user station (e.g., a server-provided application coded in Open GL™ readily executed on a Unix™—or Windows™—based system), the goal is a substantial user platform independent network interface with the server system. Security is provided through encryption and multiple firewalls at the server(s) so as to protect proprietary and sensitive data.

The accompanying drawings are included to provide a further understanding of the invention and are incorporated in and constitute part of this specification, illustrate several embodiments of the invention and, together with the description, serve to explain the principles of the invention. It will become apparent from the drawings and detailed description that other objects, advantages and benefits of the invention also exist. At the outset, it should be observed that the several primary features of the system are depicted as residing on several servers. This is, of course, only for exemplary and illustrative purposes. The features of collaborative engineering, global manufacturer's registry, and electronic bidding, can be provided via one or many servers, which may be co-located or which may reside at different locations connected through a network. Likewise, the data stored by the system could be stored at a single location or amongst multiple locations in a so-called hybrid relational object oriented database architecture. In general, the so-called first aspect of the fully integrated embodiment of the invention, the collaborative engineering feature, is described as the Network Integrated Concurrent Engineering With Computer Aided Design or NICECAD. The second aspect of the invention, the global manufacturer's registry feature, is described as the Global Manufacturer's Registry (GMR) or the Interactive System for Engineering Design and Manufacture (EDM). The so-called third aspect of the invention, the electronic bidding capability feature, is described as the Electronic Trading Community or ETC.

Additional features and advantages of the invention will be set forth in the description that follows, and in part will be apparent from the description, or may be learned by practice of the invention. The objectives and other advan-

tages of the invention will be realized and attained by the system and methods, particularly pointed out in the written description and claims hereof as well as the appended drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

The purpose and advantages of the present invention will be apparent to those of skill in the art from the following detailed description in conjunction with the appended drawings in which like reference characters are used to indicate like elements, and in which:

FIG. **1** is an diagram illustrating the concurrent engineering principle, the various teams which may be involved, and the chronology of a typical concurrent engineering development project.

FIG. **2** is a block diagram illustrating an embodiment of a NICECAD system according to the present invention including a network, NICECAD server system, database, prime contractor user systems and supplier user systems.

FIG. **3** is a block diagram, according to an embodiment of the present invention, illustrating a database including stored contracts and agreements data, account data, PDM data, electronic commerce data, materials data and other data.

FIG. **4** is a block diagram, according to an embodiment of the present invention, illustrating stored contracts and agreements data of a database.

FIG. **5** is a block diagram, according to an embodiment of the present invention, illustrating account data of a database.

FIG. **6** is a block diagram, according to an embodiment of the present invention, illustrating electronic commerce data of a database.

FIG. **7** is a block diagram, according to an embodiment of the present invention, illustrating materials data of a database.

FIG. **8** is a block diagram, according to an embodiment of the present invention, illustrating PDM data of a database in greater detail.

FIG. **9** is a block diagram, according to an embodiment of the present invention, illustrating a NICECAD server system, including its processing modules.

FIG. **10** is a block diagram, according to an embodiment of the present invention, illustrating a system administrative processing module of a NICECAD server system.

FIG. **11** is a block diagram, according to an embodiment of the present invention, illustrating a PDM processing module of a NICECAD server system.

FIG. **12** is a block diagram, according to an embodiment of the present invention, illustrating a CAD processing module of a NICECAD server system.

FIG. **13** is a block diagram, according to an embodiment of the present invention, illustrating a multimedia communications processing module of a NICECAD server system.

FIG. **14** is a block diagram, according to an embodiment of the present invention, illustrating an electronic communications processing module of a NICECAD server system in greater detail.

FIG. **15** is a block diagram, according to an embodiment of the present invention, illustrating an engineering analysis and simulation processing module of a NICECAD server system in greater detail.

FIGS. **16**A & B depict a flowchart illustrating a method, according to an embodiment of the present invention, to carry out a concurrent engineering project.

FIG. **17** is a flowchart illustrating a method, according to an embodiment of the present invention, for a design team member to create an initial design.

FIGS. **18**A & B depict a flowchart illustrating a method, according to an embodiment of the present invention, for a team member to perform engineering analysis and simulation on a part design model.

FIGS. **19**A & B depict a flowchart illustrating a method, according to an embodiment of the present invention, for quasi-real time interaction among users on the NICECAD system.

FIG. **20** is an overview diagram of an interactive EDM system according to an embodiment of the invention, including the network interface, GMR server system, GMR database, fabricator user systems and designer user systems.

FIG. **21** is a block diagram, according to an embodiment of the present invention, illustrating a GMR server system in greater detail, including the various processing modules and applications that it may comprise.

FIG. **22** is a block diagram, according to an embodiment of the present invention, illustrating the GMR database in greater detail, including the stored fabricator data, account data and other data.

FIG. **23** is a flowchart illustrating a method, according to an embodiment of the present invention, for a designer search session using an interactive EDM system.

FIG. **24** is a flowchart illustrating a method, according to an embodiment of the present invention, for a designer to retrieve reports using an interactive EDM system.

FIG. **25** is a flowchart illustrating a method, according to an embodiment of the present invention, for a fabricator session using an interactive EDM system.

FIG. **26** is a functional diagram of the Electronic Trading Community of the invention, including the GMR user, GMR provider, and manufacturing vendors.

FIG. **27** illustrates how a manufacturing vendor can evaluate a part design model hosted on a GMR graphics server.

FIG. **28** illustrates how a prime contractor and manufacturing vendor can engage in a communications session using the GMR graphics server, including the substantially simultaneous display and manipulation of the part design model.

DETAILED DESCRIPTION OF THE INVENTION

As discussed in the Summary of the Invention, the present invention is directed to providing a networked, virtual, collaborative environment for three aspects of an engineering development: (1) the design and development phase (referred to as the NICECAD or Network Integrated Concurrent Engineering Computer Aided Design); (2) the identification and evaluation of qualified fabricators or manufacturers for a design (referred to as the Global Manufacturer's Registry or GMR system or the Interactive System for Engineering Design and Manufacture (EDM), and (3) the solicitation and evaluation of requests for proposals or quotes from qualified bidders (referred to as the Electronic Trading Community or ETC).

The First Aspect of the Invention: System and Methods for Undertaking an Engineering Design and Development Effort in a Virtual and Collaborative Environment

In general, the NICECAD server system provides Product Data Management (PDM), CAD functionality, engineering analysis and simulation (EAS), multimedia communications functionality, electronic commerce capability and front-end network interface and system administration support.

FIG. **2** depicts an overview of an embodiment of a NICECAD system **100** according to the present invention,

which provides for concurrent engineering in a "virtual," collaborative environment. This environment obviates engineering data format (e.g., CAD geometry), business and other boundaries that presently impede conventional approaches to concurrent engineering. Significantly, the virtual environment provides data neutrality by maintaining part design models in a common, neutral format, and providing utilities so that users' specialized part design model software formats do not present barriers to participation in a collaborative engineering effort.

NICECAD system **100** incorporates product data management (PDM), computer aided design (CAD), engineering analysis and simulation (EAS), multimedia communications and electronic commerce (EC) so that the entire project, including its engineering and business components, may be carried out in a virtual, collaborative and secure environment.

FIG. **2** depicts an embodiment of a NICECAD system **100** according to the present invention, comprising one or more prime contractor user systems **220**; one or more supplier user systems **230**; one or more databases **210**; one or more NICECAD server systems **200**; network **260**; one or more backup data storage devices **250**; and one or more backup server hardware and software devices **240**.

Network **260** may comprise any network that allows communication amongst the components, and may encompass existing or future network technologies, such as the existing "Internet," "World Wide Web," Wide Area Network (WAN), Local Area Network (LAN), "Internet Protocol-Next Generation" (sometimes referred to as the coming "Supernet"), and any variation of packet switched networks (or other supporting data packing technologies) for permitting communication amongst user systems and servers. Regarding the Internet and like shared networks, high bandwidth systems have been developed and are becoming increasingly accessible. For example, the Abilene Network developed by University Corporation for Advanced Internet Development (UCAID) and the so-called Internet2 project are two examples of such high-capacity systems. The very high performance Backbone Network Service (vBNS) provided by MCI Corporation and used by the National Science Foundation and others is another example of such a high-capacity system.

Prime contractor user systems **220** may comprise any system capable of interfacing with network **260**. Prime contractor user systems **220** may comprise "standard" computer systems that do not require specialized hardware or software to use NICECAD system **100**. Prime contractor user systems **220** may comprise personal computers, microcomputers, minicomputers, portable electronic devices, a computer network, or any other system operable to interface with network **260** to send and receive data. Prime contractor user systems **220** may comprise computers running standard operating systems and supporting "browser" technologies for accessing and displaying data over a common network, such as personal computers with Windows NT™ and Microsoft Internet Explorer™ 5.0 or Netscape Communicator™ 4.06, an Apple Macintosh™ running MOSAIC™ web browser software, a Sun SparcStation™ running UNIX and Netscape Communicator™, and a Silicon Graphics™ UNIX-based workstation such as the SGI Octane™ running Netscape Communicator™. Prime contractor user systems **220** may comprise computer systems running a so-called "custom browser" specially-adapted for the present application. In one embodiment, such a custom browser may be provided by the NICECAD provider (e.g., NICECAD server system **200**) as a downloadable file through a standard browser. As those of skill in the art may appreciate, prime contractor user systems **220** may comprise future variations of such systems that permit the interaction over a network with a server system.

In one embodiment, prime contractor user system **220** comprises a personal computer or workstation running a standard operating system such as Windows NT, and using a standard browser such as Microsoft Internet Explorer™ 5.0 capable of interpreting HTML 4.0, XML, VRML, and running Java™ applets or like "miniapplications." In another embodiment, prime contractor user system **220** comprises a personal computer running such a standard operating system and using such a standard browser in conjunction with client-side applications which, while being semi- machine-dependent in the sense they are not executed using a browser's so-called "virtual machine," are otherwise compiled and executed using a standard operating system. Applications coded in Open GL™ by Silicon Graphics, a language for graphics applications well known to those of skill in the art, and readily supported by Unix™—and Windows™—based systems, would be just one example of such a semi- machine-dependent application running on a prime contractor user system **220**. In yet another embodiment, a server-supplied so-called custom browser application is provided which is specially-adapted for interfacing with NICECAD server system(s) **200** and any client side semi- machine-dependent applications (or, alternatively, Java™-type or Java™-like miniapplications) it provides. Such semi- machine-dependent applications may be server-provided, for example, as downloadable files through a standard or custom browser. In each manifestation of the invention, the overriding goal is to preserve a substantially platform independent network interface between user systems and the server system (e.g., NICECAD server system **200**).

Likewise, supplier user systems **230** may comprise any system that may interact with NICECAD server system **200** over network **260**. Like prime contractor user systems **230**, supplier user systems **230** may comprise any computer systems, including any of the systems listed above. In one embodiment, supplier user system **230** comprises a personal computer or workstation running a standard operating system such as Windows NT™, and using a standard browser such as Microsoft Explorer™ 5.0. As with prime contractor user systems **220**, supplier user systems **230** may comprise a personal computer using a server-provided custom browser adapted for the present application. As with computer systems used by prime contractors, supplier user systems **230** may comprise a personal computer running semi- machine-dependent applications to be compiled and executed using a standard operating system.

Backup data storage **250** comprises a system for backing up the data stored by the NICECAD system **100**. Backup data storage **250** may comprise several backup technologies for redundancy. Backup data storage **250** may include tape media, CD-ROM, zip drives, optical disks or any other reliable means for backing up the data maintained by NICECAD server system **200**. Generally, backup data storage **250** may be data that is stored locally with NICECAD server system **200** or remotely. Furthermore, NICECAD server system **200**, databases **210** and backup data storage **250** may comprise part of a distributed database system. In one embodiment, backup data storage **250** may comprise redundant storage onto high capacity tape and recordable CD-ROM.

Backup server hardware and software **240** comprises one or more backup servers, including server modules, to reli-

ably support NICECAD server system **200** to minimize the impact of "crashes" and other events otherwise interfering with smooth operation. In one embodiment, at least one redundant server may be provided to substitute for NICECAD server system **200** should it fail. Generally, backup server hardware and software **240** may reside locally with NICECAD server system **200** (or a part of NICECAD server system **200**, if it is distributed across several different servers, to be discussed below), although it may be physically remote, and may even be accessible through a different network address (e.g., at a different URL or web site).

Databases **210** depict the storage media that may be employed to store data maintained by the NICECAD system. Databases **210** may be one or more physically distinct media, including, but not limited to, hard drives, floppy drives, CD-ROM, and any other existing or future storage technologies supporting ready access. Databases **210** may store information for a concurrent engineering development project, such as contracts data, engineering data, account data and other project related data (further discussed below in conjunction with FIGS. **3–8**). In one embodiment, databases **210** resides locally with NICECAD server system **200**. In another embodiment, databases **210** is remotely located from NICECAD server system **200**.

NICECAD server system **200** comprises a server system supporting the interactive collaborative engineering environment. NICECAD server system **200** interfaces with prime contractor user systems **220** and supplier user systems **230** through network **260**. NICECAD server system **200** may include the hardware and software to support and interface with prime contractor and supplier user systems on a substantially platform independent basis. This minimizes user systems' requirements for specialized hardware or software capabilities. This substantial platform independence arises from the fact that NICECAD server system **200** provides certain specialized applications as either client-side machine-independent applications, such as Java™ applets or the like, executable using a standard browser (or by a custom browser supporting such applications), and/or as client-side semi- machine-dependent applications, such as those coded in Open GL™ or like coding tools, to be executed on a machine with a standard operating system. Ultimately, significant benefits of the invention derive from the fact that the user system interface with NICECAD server system **200** is substantially platform independent.

Generally, NICECAD server system **200** includes system administration and network-related software modules, as well as the various specialized software modules for CAD, EAS, multimedia communications and EC. Certain of the software modules may be client-side interactive for supporting specialized concurrent engineering tasks. For example, NICECAD server system **200** may also include the hardware and software for multimedia operations such as audio, video and graphics, that may be transmitted over the network for presentation to user systems with standard client-side multimedia support.

In one embodiment, NICECAD server system **200** may be publicly accessible as a web site, but may provide multiple levels of security (e.g., multiple "firewalls") to ensure that proprietary project data may be protected. For example, NICECAD server system **200** may be accessed only by users with a user ID and password. In one embodiment, the data transmitted is compressed and encrypted using encryption at least as reliable as that provided by RSA 1024 bit encryption keys, a technology well known to those of skill in the art. In one embodiment, NICECAD server system **200** may comprise a "back end" processing server running UNIX™ for the various database and specialized processing operations, and a "front end" web server running Windows NT™ and Microsoft's Transaction Server™ for network-related operations.

As illustrated in FIG. **2**, the architecture of NICECAD server **200** may be distributed. In other words, hardware and software for the various NICECAD system functions is not necessarily resident on one server system at one physical location. Thus, in one embodiment, as depicted in FIG. **2**, NICECAD server system **200** comprises several systems for hosting software for various functions, such as PDM server system **202** for product data management functionality; CAD/EAS server system **204** for computer aided design and engineering analysis and simulation; MM/EC server system **206** for multimedia communications support and electronic commerce; and network interface/system administration **208** for system administration and network related tasks. Moreover, these distributed systems may interface through network **265**, which may comprise any of the network technologies discussed in connection with network **260**. Of course, those of ordinary skill can appreciate that this particular distributed architecture is exemplary, and the particular embodiment of FIG. **2** is intended to express the general principle that all of the NICECAD server system **200** functionality need not be co-resident. In one embodiment, where the NICECAD system functions are distributed in this fashion, a single network address, such as an Internet address or URL or web site address, is used by the user systems. Access to other parts of the NICECAD system through network **265** is automated so that the system appears transparent or "seamless" to the user system.

The Database used by the System for Engineering Development

FIG. **3** illustrates database(s) **210** maintained by NICECAD system **100**, which may comprise contracts and agreements data **305**; account data **320**; materials data **331**; electronic commerce data **394**; PDM data **335**; and optionally other data modules **395**.

Contracts and agreements data **305** may comprise information stored by NICECAD server system **200** relating to contracts or agreements among the various actors. This data may comprise contracts between users and NICECAD system data module **410** and contracts between prime contractors and suppliers data module **415**, as depicted in FIG. **4**. The former may generally comprise those agreements regarding the terms of usage of the NICECAD system, such as fee structure, security, assignment of liability, and so on. The latter may contain records of contracts or agreements entered into between prime contractors and suppliers, or between suppliers. In one embodiment, such contracts may be prepared over the network by tailoring standard contracts and contract terms provided by the NICECAD system, further discussed in connection with FIG. **6**.

Account data **320** may comprise information stored by the NICECAD system pertaining to use by prime contractors and suppliers. For example, the information may comprise records reflecting all user transactions, including billing information, so that a complete history associated with a particular project is available. In one embodiment, every concurrent engineering development project is assigned an identifier, such as project number or project name, which may be used to retrieve account records in a relational database arrangement. In one embodiment, account data **320** may be logically divided into supplier account data module **525** and prime contractor account data module **530**, as illustrated in FIG. **5**.

Electronic commerece (EC) data **394** may comprise data related to electronic commerece transactions carried out

using NICECAD system **100**. The NICECAD server system **200** may be configured to allow business transactions pertaining to the engineering development effort to be carried out in the virtual environment, such as negotiations, contracting, and funds tranfer. EC data **394** may be used to store records of such transactions, and may comprise stored standard contracts data module **696**; stored standard terms and conditions data module **697**; stored electronic business documents data module **698**; and stored EC digital signature data module **699**, as illustrated in FIG. **6**.

Stored standard contracts data module **696** may comprise a series of contract "templates" for prime contractors and suppliers to use as a starting point for creating an agreement. For example, in one embodiment there is a standard form agreement for a fabricator to produce a quantity of prototypes of a design within some time-frame. There may be another standard form contract for an EAS team to perform some specified analysis on a design. Stored standard terms and conditions data module **697** may comprise various terms and conditions to be used in preparing an agreement including such things as liquidated damages, arbitration clauses, procedures for engineering changes, and the like. Those of ordinary skill can appreciate that module **696** may be used with or without module **697**, and that said modules could easily be combined.

Stored electronic business documents data module **698** may comprise documents or records related to all EC transactions. For example, when a prime contractor transfers funds to a supplier, a record may be stored in this module. If standard form contract or standard terms and conditions data is accessed, a record may be stored. If a contract is actually entered into using the NICECAD system **100**, a record may be stored. In general, this module ensures that NICECAD system **100** provides an electronic "paper trail" that provides a history of the business transactions carried out using the NICECAD system.

Stored EC digital signature data module **699** may comprise data of a "digital signature" required to formalize agreements between users. Analogous to the usual process, an agreement may be considered finalized when the parties provide their digital signature signifying their assent to the terms.

Materials data **331** may contain information relating to materials that may be used to fabricate a part design model and may be used as a reference tool for designers, analysts and fabricators. Materials data **331** may contain property data as well as applications data. For example, materials data **331** may have an entry for a titanium alloy identifying properties such as elastic modulus, tensile strength, hardness, machinability, ductility or other properties. Materials data **331** might contain applications data for the titanium alloy, such as its limitations when used in a high oxygen environment. As illustrated in FIG. **7**, materials data **331** may be logically subdivided into homogeneous materials data module **732** for homogeneous materials; heterogeneous materials data module **733** for heterogeneous materials; and other materials data module **734** for materials not easily classified. For example, properties and/or applications data for materials like metals, plastics, ceramics and the like, may be stored in module **732**. Properties and/or applications data for materials like carbon fiber-epoxy or Kevlar™ composites may be stored in module **733**.

Materials data **331** may be used as a reference source or it may be used in conjunction with the CAD tools to create a design. For example, materials data **331** may be used by a design team to assign or associate certain materials with entities of a part design model (such as a 3D solid model). Subsequently, a fabricator team analyzing the 3D solid model (e.g., when analyzing the design to prepare a bid) may ascertain that a particular material is to be used for a particular entity, and then may learn about applications and properties of that material by accessing materials data **331**.

PDM data **335** may contain the data stored for individual concurrent engineering development projects. Broadly speaking, the concept of PDM is to create an information infrastructure for the entire history of an engineering effort, including technical data (such as the design evolution and analysis/simulation results), as well as business or management data (such as contracts and budget/schedule information). Accordingly, PDM data **335** may include modules with project specific data (such as part design models for particular projects) and non- project specific data (such as reference modules to facilitate creating part design models). PDM data **335** may comprise the following modules: stored product data management system electronic document data module **840**; stored product data and electronic document distribution control module **855**; stored design and analysis access permission data module **860**; stored baseline part design model data module **865**; and stored working copy part design model data module **892**; standard and custom parts library data module **875**; stored engineering analysis and simulation results data module **885**; manufacturing standards and specifications data module **850**; stored standard attibutes and attribute values data module **845**; stored **880**; stored quasi-real time multimedia communications sessions data module **890**; and other PDM data modules **891**, as illustrated in FIG. **8**.

Stored product data management system electronic document data module **840** may comprise non-graphic project documents, such as project specifications, change documents, revision history documents, and budget and schedule documents. Change documents refers to any documents that may be prepared to implement a design change. For example, in Department of Defense applications, such documents are sometimes referred to as "engineering change proposals" or "engineering change orders." Revision history documents refers to any documents that list or provide a history of design changes.

Stored product data and electronic document distribution control data module **855** may comprise a module for maintaining "check-in/check-out" records that document when team members access certain documents in NICECAD system **100**. In one embodiment, every time a user accesses or "checks out" a specification or part design model, a "time-stamp" and team member identifier is stored in this module (e.g., see FIG. **11**, module **1104**). This serves configuration management by providing an electronic "paper trail."

Check-in/check-out controls generally refers to the procedures employed by the NICECAD system to control access to proprietary part design model and specification data. As will be discussed further below, the prime contractor may assign access permissions to part or all of the part design model, project specification, and the EAS processing modules. This ensures that the prime contractor has control over which teams access which data, and which teams run which analysis. Whenever an authorized team member accesses the part design model or specification, the team member is said to have "checked out" that item. Once the team member completes the task, he/she may have to "check in" the item by informing the NICECAD system that the task is complete. By updating the check-in/check-out data in stored product data and electronic document distribution control data module **855**, the NICECAD system provides

configuration control by maintaining a history of which teams have accessed which part design models and documents.

It is important to appreciate that check-in/check-out records may still be maintained even if an approval authority does not limit access to the part design model, specifications, or EAS processing modules.

Stored design and analysis access permission data module **860** may comprise data assigned by the prime contractor determining which teams (or team members) may access the part design model, documents and EAS processing modules. The module may also comprise data determining which teams may access certain project documents, such as specifications. A part design model generally comprises a series of geometric and topological entities. Teams, such as EAS teams, may need access to all or part of a part design model in order to carry out the analysis for their specific discipline. They may need to access specifications or other documents to perform their tasks. Likewise, those teams may need to access one or more EAS processing modules to carry out the analysis. Stored design and analysis access permission data module **860** allows an approval authority to assign access permissions to limit access to those portions of the part design module, those specifications (or portions thereof), and those EAS processing modules as appropriate. This serves configuration control by limiting access to only those who need it.

Stored baseline part design model data module **865** may contain the current approved version of the design referred to as the "baseline." Each time a design change is approved by the prime contractor, the baseline part design model may change. A part design model may be created using the NICECAD CAD capability, and generally comprises a series of entities (e.g., topological or geometric features) which may be assigned attributes and attribute values. Attributes include such items as tolerance, surface finish, material, special fabrication instructions, etc. In general, stored baseline part design model data module **865** may contain the part design models, such as 3D solid models, including attributes, for the projects in NICECAD system **100**. In one embodiment, this module may contain the entire history of baseline designs, each of which is assigned a version number for tracking purposes (e.g., Design **1.00**, **1.01**, **1.02**, etc.). The latest version number may be the current baseline design.

Stored working copy part design model data module **892** may be used by designers and analysts as a virtual "scratch pad" for storing part design models. For example, an EAS team member who checks out the current baseline part design model from module **865** may not be permitted to "check in" that part design model. This is because it may be that only the prime contractor can authorize writing a baseline part design model to module **865**. This provides configuration control and protects the integrity of the current baseline part design model. However, the EAS team member may use stored working copy part design model data module **892** to store a "working copy" of the part design model. For example, if a fabricator team analyzing producibility determines that certain design changes should be effected, a proposed revised baseline part design model may be stored in this module. Project team members may also use this module to temporarily store working copies of the baseline part design model while they are completing their analysis. This way the team member does not have to go through the check-out procedures repeatedly while performing the same analysis. In one embodiment, the working copies stored in this module by NICECAD server **200** are assigned working copy version numbers. In another embodiment, the working copy version numbers are maintained so that there is a complete history of all working copies of the part design model in the NICECAD system.

Standard and custom parts library data module **875** may comprise a library of "pre-built" standard and custom parts that may be accessed by designers to include in larger part design models. Availability of such data may expedite the design creation and modification process (e.g., see FIG. **9**, module **932**). Standard parts may include such common items as fasteners, gears, bearing sets and the like. Custom parts may include parts previously built by prime contractors or suppliers that are saved for use in later projects. For example, a sonobuoy manufacturer may save a series of custom parts for various configurations of sound-detecting underwater hydrophones.

Stored engineering analysis and simulation results data module **885** may comprise data stored by the NICECAD system reflecting the results of engineering analysis performed by EAS teams. The EAS analysis may be performed using internal NICECAD processing modules (e.g., see FIG. **9**, module **946**) or external processing modules. Stored engineering analysis and simulation results data module **885** ensures that the virtual NICECAD collaborative engineering environment includes a record of analysis performed on a part design model. Those of ordinary skill should appreciate that the term "engineering analysis and simulation" is generally descriptive of studies performed by various teams considering a design. The term is not meant to suggest that analysis and simulation are discrete tasks or disciplines. Some studies may involve only analysis of a design; some may involve only simulation of the operation or manufacture of a design; and some studies may require both simulation and analysis.

Manufacturing standards and specifications data module **850** may comprise fabrication standards, inspection standards and other standards. For example, to support Department of Defense applications, this module may contain various MIL-STD or MIL-SPEC documents defining such things as manufacturing standards, inspection standards, reliability, quality assurance, safety, and so on. This module may be used as a reference source providing users the text of the standards and specifications. This module may be used by designers to associate particular standards with graphical entities of a part design model. In one embodiment, a design team member may associate a standard (e.g., MIL-STD-5556.8) with a particular graphical entity when creating a part design model. When viewing the design using a CAD tool (e.g., see FIG. **9**, module **932**) a fabricator team member may see the standard associated with a particular entity that may be "clicked on" to link to the text of the standard.

Stored standard attributes and attribute values data module **845** may comprise a series of standard attributes and attribute values for use by a designer in creating a design. As previously mentioned, a part design model may comprise a series of geometrical and topological entities. Each entity may be assigned its own attributes, such as surface finish, material, tolerance, and the like. For some attributes, each attribute is given a value or category. For example, tolerance may be given a tolerance type (e.g., flatness, linear, angular, runout, etc.) and a tolerance range. For surface finish, a surface finish value is assigned. Stored standard attributes and attribute values data module **845** may be used in connection with a CAD tool (e.g., see FIG. **9**, module **932**) so that a designer may rapidly create an initial design including the assignment of attributes to the part design model entities.

Stored PDM digital signature data module **870** may comprise digital signatures, well known to those of skill in the art, by the "approving authority" for the part design model. In conventional collaborative engineering environments, the "approving authority" (usually the prime contractor) signifies approval of the baseline part design model and associated specification documents by affixing a written signature. In the virtual NICECAD environment, a digital signature may be associated or included to certify approval of certain items, such as the approved current baseline part design model, the approved final part design model submitted for fabrication, and various supporting specification documents. This digital signature feature aids configuration control by providing a quick and effective way to determine whether a particular electronic file represents an "approved" version or is a mere working copy (or proposed redesign). In one embodiment, design changes require a digital signature by the approving authority, whereby a new design must include the proper digital signature (e.g., a key) before it may be stored in stored baseline part design model data module **865**. In other embodiments, digital signatures may not be required.

Stored standard drawing symbols data module **880** may comprise a series of standard drawing symbols, such as ASME Y14.5 symbols. These symbols may be used by a designer in creating a part design model using the CAD tool (e.g., see FIG. **9**, module **932**). In one embodiment, where the CAD tool includes a custom design visualization functionality (e.g., see FIG. **12**, module **1210**, discussed below), stored standard drawing symbols data module **880** may be automatically accessed by NICECAD server system **200** to provide the proper drawing symbols for a two-dimensional sectional view of a 3D part design model.

Stored quasi-real time multimedia communications sessions data module **890** may comprise records data of multimedia communications sessions (e.g., see FIG. **9**, module **978**) between teams members in a concurrent engineering development project. For example, if a design team and EAS team have a multimedia communications session using the NICECAD system to discuss certain design issues, a record may be stored reflecting the session. In one embodiment, data reflecting the actual transactions (such as audio, video and graphics) may be stored for a session. In another embodiment, memory resources are preserved by storing only descriptive information, such as the identity of the parties, duration of the session, part design model version numbers accessed and the like.

Other PDM data modules **891** may comprise any other data modules in PDM data **335**. Other data modules **395** may comprise any other data modules in database **210**.

The NICECAD Server System

FIG. **9** illustrates NICECAD server system **200**, which may comprise various software components or modules supporting processing functions. At the outset, it should be noted that in one embodiment, user access to the processing modules of FIGS. **9–15** may be via platform independent graphical user interfaces (e.g., browser pages) presented to the user's standard or custom browser. In another embodiment, user interface to processing modules may be further enabled by the provision of platform independent miniapplications (e.g., Java™ applets or the like) that may be run on user stations. Such applications may reduce or eliminate the need for specialized hardware/software at user stations, may provide client/server interactivity, and may be used to reallocate processing burdens between the server system and user systems. For example, applications involving 3D graphics presented at user stations and/or the transmission of 3D graphics between user stations and NICECAD server system **200** can be implemented using a high-level platform independent 3D graphics programming tool such as Java 3D™ 1.1 API.

As noted previously, in one embodiment user access to the processing modules of NICECAD server system **200** is facilitated using a custom browser specially adapted for the present application. Additionally, certain client-side applications are better provided as semi- machine-dependent applications that are compiled and executed using a standard operating system, although not via the "virtual machine" provided by a standard browser. For example, certain graphics-intensive applications, such as those for supporting the invention's virtual CAD and multimedia video/audio/graphics capability, ay be provided as Open GL™-coded applications rather than Java 3D™-coded applets.

It bears reiteration that NICECAD Server System **200**, primarily for design development, and GMR Server System **1000** (discussed below), primarily for locating qualified fabricators, and GMR Graphics Server **2710** (discussed below), primarily for the bidding process, may exist at the same location on the same server system. In the preferred embodiment, the overall system provides a comprehensive functionality, hosted on a single system, for the entire product development lifecycle, from initial idea by a prime contractor/designer to production in quantity by an outsourced fabricator. For illustrative purposes, we describe the various functional components as residing on separate servers, although this architecture is not required, and in the preferred embodiment, the three aspects of the invention are integrated into a substantially "seamless" virtual environment.

As illustrated by FIG. **9**, NICECAD server system **200** may comprise system administration processing module **902**; PDM processing module **918**; CAD processing module **932**; multimedia communications processing module **978**; electronic commerce processing module **988**; and engineering analysis and simulation processing module **946**.

System Administrative Processing in the NICECAD System

System administration processing module **902** generally supports system administrative processing and network related processing, and, as depicted in FIG. **10**, may further comprise user logon authentication processing module **1002**; contracts processing module **1004**; account data and other report generation module **1006**; security processing module **1008**; network interface processing module **1012**; data backup and archiving processing module **1013**; and other application modules **1014**.

User logon authentication processing module **1002** may comprise a security module for limiting access to the NICECAD system. User logon authentication module **1002** may enable a user to enter a user ID and password to log onto NICECAD system **100**. Other security modules may also be provided. In one embodiment, a prime contractor initiates a development project on NICECAD system **100** and accordingly identifies the various teams (and team members). This module then gives the various team members authorization to log onto the system for a given project.

Contracts processing module **1004** may comprise a module that ensures that before a prime contractor or supplier uses NICECAD system **100**, contracts specifying the responsibilities of each party regarding data protection, maintenance and system usage are formalized (e.g., see FIG. **4**, module **410**). In one embodiment, contracts processing module **1004** includes a platform independent client-side Java™ applet (or like miniapplication) that may be launched at user stations to facilitate the contract execution process.

Account data and other report generation module **1006** may comprise a module that formats and sends reports requested by users that, in one embodiment, may be sent as browser pages or links to downloadable files. This module may prepare reports of records of transactions on the NICECAD system or billing summaries. (See, e.g., FIG. **5**, module **320**). In one embodiment, a prime contractor may retrieve reports pertaining to all transactions relating to a given project, while suppliers may retrieve reports only pertaining to their own activity.

Security processing module **1008** may comprise a module for providing security for NICECAD system **100**. It is important that project data and multimedia communications sessions are secure, so this module may provide multiple firewalls, encryption, hashing, or other known information technology security techniques to protect data and communications. In general, project data may be treated as proprietary to the prime contractor, and security processing module **1008** provides the proper safeguards. In one embodiment, security processing module **1008** may comprise a software component using Java™ Cryptography Extension (JCE) 1.2 and Java™ 2 for encryption and/or message authentication. In the preferred embodiment, data is compressed and also encrypted using technology at least as reliable as that provided by RSA 1024 bit encryption keys.

Network interface processing module **1012** may comprise a module for supporting the interface with user systems over network **260**, or for interface with other parts of NICECAD server system **200** via network **265**. In one embodiment, network interface processing module **1012** supports the interface with user systems on a substantially platform independent basis using browser pages or the like. In this manner, user interface with NICECAD server system **200** is achieved using more or less standard hardware and software. As previously noted, such browser pages may be configured to interface with a so-called standard, commercially available browser (e.g., Netscape™ or Microsoft Explorer™) or a custom browser specially adapted to support the graphics-intensive networked CAD and multimedia communications aspects of the present invention.

Data backup and archiving processing module **1013** may comprise a module supporting periodic backing up and archiving of data (e.g., see FIG. **2**, module **250**). This module may also support the resort to backup server hardware and/or software (e.g., see FIG. **2**, module **240**) when there are system crashes or other interruptions in availability of the primary server hardware and software (e.g., see FIG. **2**, module **200**).

Other application modules **1014** may comprise any other application modules run by NICECAD server system **200** to support system administration related tasks.

Product Data Management Processing in the NICECAD System

PDM processing module **918** provides the processing associated with the NICECAD system's PDM functionality, and, as depicted in FIG. **11**, may include PDM electronic document processing module **1102**; product data and electronic document distribution control module **1104**; baseline part design model management module **1106**; working copy part design model management module **1108**; attributes and attribute values management module **1110**; PDM digital signature processing module **1112**; engineering analysis and simulation results management module **1114**; design and analysis access permissions management module **1116**; and other PDM processing modules **1118**.

PDM electronic document processing module **1102** may comprise a module that implements the general PDM capability of NICECAD system **100**. This module may provide for the data management and configuration control of non-graphic project documents such as project specifications, change documents or revision history documents (e.g., see FIG. **8**, module **840**). In one embodiment, PDM electronic document processing module **1102** comprises a substantially COTS (commercial off-the-shelf) software package tailored for NICECAD system **100**.

Product data and electronic document distribution control module **1104** carries out the check-out/check-in management procedures of the NICECAD system (e.g., see FIG. **8**, module **855**). As mentioned previously, the check-in/check-out procedures ensure the integrity of the part design model and associated specification documents. This module may provide that each time a controlled item, such as a part design model or a specification document, is accessed from the NICECAD system, a record is stored reflecting that transaction. In one embodiment, this record may comprise a time-stamp and user ID.

Baseline part design model management module **1106** may be a module that provides data management of the baseline part design model. In general, baseline part design model management module **1106** may be integrated with or may cooperate with a CAD processing module (e.g., see FIG. **9**, module **932**) so that the latter provides the substantive CAD capability and the former provides the PDM functionality. Baseline part design model management module **1106** may be integrated with a data module such as that of FIG. **8**, module **865**, to store the baseline part design model. In one embodiment, configuration control is furthered by having the NICECAD server system **200** configured so that baseline part design model management module **1106** permits only one category of user, such as approval authority users, to create or make changes to the baseline part design model. This module (**1106**) may cooperate with product data and electronic document distribution control module **1104** to enforce the check-in/check-out procedures. This module (**1106**) may provide that each version of the baseline part design model is given a version number.

Working copy part design model management module **1108** is similar to baseline part design model management module **1106**, except that the former manages the data associated with so-called "working copies" of the part design model (e.g., see FIG. **8**, module **892**). This module may provide PDM management when cooperating with a CAD processing module that provides substantive CAD processing to a user relying on a working copy. This module may provide that each version of a working copy part design model is given a version number.

Attributes and attribute values management module **1110** may comprise a module that provides PDM management of attributes and attribute values (e.g., see FIG. **8**, module **845**) inputted by a user. This module may provide that attributes and attribute values assigned to a part design model be stored with the part design model (e.g., see FIG. **8**, modules **865** and **892**) or in a separate location. This module may cooperate with a CAD processing module (e.g., see FIG. **12**, module **932**) that permits a user to view the part design model and assign attributes and attribute values to selected entities. In one embodiment, this module cooperates with a platform independent client-side application (such as a Java™ applet running on a browser) or a semi- machine-dependent application that permits the user to view the part design model and assign attributes and attribute values by selecting entities with a mouse or like device.

PDM digital signature processing module **1112** may comprise a module for managing the digital signature data in

NICECAD system **100**. In one embodiment, this module provides the user a platform independent application, such as a Java™ applet, that permits the user to append or associate a digital signature with a part design model or document (such as a project specification). The digital signature component (e.g., the key) known to the user is uploaded to the NICECAD system, where PDM digital signature processing module **1112** ensures that it is properly validated and that appropriate records of the signing are made (e.g., see FIG. **8**, module **870**). In one embodiment, PDM digital signature processing module **1112** may comprise a software component using Java™ Cryptography Extension (JCE) 1.2 and Java™ 2 for creating the digital signatures.

Engineering analysis and simulation results management module **1114** may be a module that manages the EAS results for the NICECAD system (e.g., see FIG. **8**, module **885**). This module may cooperate with modules that perform the substantive EAS processing (e.g., see FIG. **15**, module **946**).

Design and analysis access permissions management module **1116** may comprise a module that provides the processing to allow a prime contractor or approval authority to assign access permissions to various teams or team members. This module may provide the processing to determine whether access is granted or denied when a data request is made. In one embodiment, NICECAD server system **200** facilitates the access assignment process by providing a platform independent interactive application, such as a Java™ applet, that lists various project data (such as the part design model and project specifications) and EAS processing modules, and the various team members. The prime contractor, through the use of keyboard input, "mouse," or similar device, may then assign viewing permissions for each item, which are then uploaded to the NICECAD server system **200** (e.g., see FIG. **8**, module **860**).

Other PDM processing modules **1118** refers to any other PDM processing modules for managing the data in a concurrent engineering development project.

Computer Aided Design (CAD) Processing

At the outset, it should be appreciated that CAD processing module **932** may be used by design and analysis team members during the design and development phase to create, analyze and modify a part design model represented as a three dimensional solid model file or the like. CAD processing module **932** may also be linked to the second aspect of the overall system (GMR database searching for qualified fabricators) insofar it provides a means to create and/or upload models for a part design that will be analyzed using the producibility logic to be discussed below in connection with FIGS. **20–25**. Alternatively, the conversion utility of CAD processing module **932** could be used to upload and convert for storage in a neutral format a part design model already in existence (i.e., not initially created using the NICECAD collaborative, virtual environment). CAD processing module **932** may also be linked to the third aspect of the overall system (Electronic Trading Community) insofar it provides a means to create (or upload and convert a previously created part design model) and store a part design model that will be made available to fabricators preparing bids for a fabrication effort.

As previously discussed, computation- and graphics-intensive applications, such as the present inventions CAD functionality, may be provided according to several possible embodiments, each of which pursues the goal of a substantially independent network interface between user systems and a central server system. For example, in one embodiment, CAD processing module **932** and user systems (e.g., prime contractor user systems **220** or supplier user systems **230**) interface through standard or server-provided custom browsers with 3D part design model graphics presentation and manipulation facilitated using server-supplied (e.g., by CAD processing module **932**) Java™ applets (or like miniapplications which can be executed on a browser). In another embodiment, so-called semi- machine-dependent applications may be provided by CAD processing module for execution on a user system with a standard operating system to support demanding graphics applications.

CAD processing module **932** may be a module for supporting the CAD processing capability of NICECAD system **100** and, as depicted in FIG. **12**, may further comprise part design model creation and editing processing module **1202**; standard and custom parts library management module **1204**; standard drawing symbols management module **1206**; manufacturing standards and specifications management module **1208**; part design model visualization and manipulation processing module **1210**; CAD utilities conversion/plotting module **1214**; and other CAD support processing modules **1216**.

Part design model creating and editing processing module **1202** is a module that may provide the capability for a user to create and modify part design models, such as 3D solid models. In one embodiment, this module includes a platform independent client side application, such as a Java™ applet, so that the user may rely on standard hardware and software, including a standard graphical interface (such as a web browser), in order to create and modify part design models. In one embodiment, part design model creating and editing processing module **1202** comprises a commercially available software product, such as Mechanical Desktop™ from AutoDesk, SolidEdge™ from Unigraphics, or Quick HDL or Autologic HDL™ from Mentor Graphics Corp., that has been customized and implemented for a server-based networked application. In one embodiment, the provision of graphics (such as 3D graphics) over network **260** is implemented using graphics applications known in the art, such as Java 3D™ 1.1 API (Application Programming Interface), which can be used for creating and manipulating geometry of 3D graphical objects. As previously noted, another embodiment provides that particularly demanding graphics applications be provided as semi- machine-dependent applications, such as those coded in Open GL™, provided to user systems for execution. As before, the overriding goal is to maintain a substantially user platform independent network interface, preserve data neutrality, and focus the processing (computational) burden on the server system rather than the user systems.

Standard and custom parts library management module **1204** may be a module that coordinates the use of a library of standard and custom parts by a team member creating a design (e.g., see FIG. **8**, module **875**). In one embodiment, this module represents a custom feature added to or integrated with a commercially available CAD product, as discussed above.

Standard drawing symbols management module **1206** may be a module that coordinates the use of a library of standard drawing symbols by a team member creating a design (e.g., see FIG. **8**, module **880**). In one embodiment, this module represents a custom feature added to or integrated with a commercially available CAD product, as discussed above. This module may also associate the proper drawing symbols with entities of a part design model that is depicted as a two-dimensional section after being subjected to the "virtual cutting tool" (discussed further below in connection with module **1210**).

Manufacturing standards and specifications management module **1208** may be a module that coordinates the use of a library of standards and specifications by a team member creating a design (e.g., see FIG. **8**, module **850**). In one embodiment, this module represents a custom feature added to or integrated with a commercially available CAD product, as discussed above. For example, the embodiment of part design model creation and editing processing module **1202** using a Java™ applet or the like may be linked to manufacturing standards and specifications management module **1208**. In this embodiment, a user creating a part design model can readily assign standards and specifications to graphical entities. In another embodiment, this module permits a user viewing a design (such as a fabricator evaluating a design in order to develop a bid) to link from an attribute associated with an entity to the underlying specification or standard.

Part design model visualization and manipulation processing module **1210** may be a module that permits users to manipulate (e.g., translate, rotate, etc.), "mark up" and "cut" a part design model, such as a 3D solid model. This capability may be considered a supplement to the basic CAD functionality of part design model creation and editing processing module **1202**. "Mark up" refers to comments or marks that may be appended to the part design model, which may be especially usefull when engaged in an interactive multimedia session (see FIG. **19**, below). Part design model visualization and manipulation processing module **1210** may permit software "cuts" using a "virtual cutting tool" to provide two-dimensional (2D) sections of a 3D part design model. The 2D sections may be appended with drawing symbols from a library of drawing symbols (e.g., see FIG. **12**, module **1206**, and FIG. **8**, module **880**) so that the electronic equivalent of standard 2D drawings are generated. Part design model visualization and manipulation processing module **1210** also supports functionality to view the design according to different perspectives, including zoom and pan functions, and so-called virtual "fly throughs." "Fly throughs" allow the user to inspect interior regions of a part design not readily ascertainable in standard drawing perspectives.

In one embodiment, part design model visualization and manipulation processing module **1210** includes a platform independent client application, such as a Java™ applet, that is integrated with the NICECAD system's multimedia communications feature (e.g., see FIG. **13**, module **978**) so that single or multiple "on line" users may view 2D sectionals, fly throughs, and other views of the part design model that are generated in a substantially "on the fly" fashion. In one embodiment, part design model visualization and manipulation processing module **1210** comprises an application coded in Java 3D™ 1.1 for 3D graphics. In this embodiment, the part design model is presented as a Java 3D™-converted representation of the part design model stored in a neutral format, such as AP214 STEP format. In another embodiment, part design model visualization and manipulation processing module **1210** comprises a server-provided semi- machine-dependent application (such as one coded in Open GL™) for execution on user systems with standard operating systems.

CAD utilities conversion/plotting module **1214** may be a module that provides various CAD utilities, such as file format conversion and output plotting. For example, this module may permit users to convert part design models among various formats, such as AutoCAD™, ProEngineer™, IGES, STEP, and other CAD file formats known to those of skill in the art. This capability may be useful when an EAS team member is performing an "off-line" analysis requiring a format different from that normally stored in NICECAD system **100**. This module may permit a user to generate electronic or hard copies of the part design model, such as 2D drawings generated using the cutting tool. In one embodiment, CAD utilities conversion/plotting module **1214** includes a platform independent Java™ applet or the like to facilitate file conversion or plotting. In the preferred embodiment, this platform independent application provides data compression functionality to reduce the time required for uploading in bandwidth-constrained environments. The corresponding software component at CAD processing module **932** supports decompression to reconstitute the original files.

As with other capabilities provided by the NICECAD system, CAD utilities/conversion plotting module has applications to all three aspects of the overall system (NICECAD, GMR and ETC). Therefore, it may be used by a designer to upload and convert an already-existent part design model file onto the system before conducting a search of the GMR database for qualified fabricators. It may be used by a designer or prime contractor (or by fabricators) to upload and/or convert part design model files when engaging in the bidding process.

Other CAD support processing modules **1216** may comprise any other processing modules supporting the NICECAD system's CAD functionality.

Multimedia Communications Processing

The multimedia communications functionality has applications to all three of the aspects of the present system. Therefore, the multimedia communications capability may be used by design team members when developing and evaluating a design. It may be used by a designer or prime contractor to engage in quasi- real-time discussions with potential fabricators regarding design and/or contractual issues.

Multimedia communications processing module **978** provides the multimedia communications capability of the system and, as depicted in FIG. **13**, may comprise quasi-real time audio processing module **1302**; quasi-real time video processing module **1304**; quasi-real time graphics processing module **1306**; and other multimedia processing modules **1308**.

Quasi-real time audio processing module **1302** permits users to engage in an "on-line" interactive communications session through NICECAD server system **200**. In one embodiment, the system provides a secure environment so that users may engage in oral communications using quasi-real time audio processing module **1302** without concern about disclosing proprietary or sensitive information.

Quasi-real time video processing module **1304** permits users to engage in a NICECAD communications session that includes video to thereby conduct a secure teleconference-like session through NICECAD server system **200** over network **260**.

In one embodiment, the communications support provided by quasi-real time audio processing module **1302** and quasi-real time video processing module **1304** are coordinated so that video and audio are presented to each participant in a substantially synchronized manner. In one example, the NICECAD system's capability for transmission of audio and video is implemented using a software tool for synchronization, control, processing and presentation, such as Java™ Media Framework (JMF) 1.0 API or JMF 2.0.

Quasi-real time graphics processing module **1306** is a module that may permit participants in an on-line commu-

nications session to view graphics, such as a 3D part design model or 2D section cuts, rotations, fly-throughs, zooms or pans in a substantially concurrent manner. In one embodiment, this module may be linked or coordinated with a module such as part design model creation and editing processing module **1202** (which may include a server-provided Java™ applet or semi- machine-dependent application) or part design model visualization and manipulation processing module **1210** (which may include a Java™ applet). This embodiment provides a substantially real time on-line session for creation and editing of a part design model and/or manipulation, translation and "virtual cuts" of a part design model. This would allow, for example, one design team member to engage in the design effort with another remotely located design team member. This would allow, for example, a design team member to interact with a remotely located fabricator team member to view sectional cuts at the same time in order to discuss producibility issues. In one embodiment of the invention, this functionality of the NICECAD system is provided using applications coded in Java 3D™ 1.1 API for supporting the transmission of graphics over network **260**.

Other multimedia processing modules **1308** comprises any other modules supporting the NICECAD system's multimedia communications capability.

Electronic Commerce Processing

Electronic commerce processing module **988** provides the electronic commerce capability of the system and, as depicted in FIG. **14**, may comprise standard contracts processing module **1402**; terms and conditions negotiation processing **1404**; contract formalization processing module **1406**; electronic finds transfer processing module **1408**; and other electronic commerce processing modules **1410**.

Standard contracts processing module **1402** may be a module that permits users (such as a prime contractor and a supplier or fabricator) to access standard form contract templates (e.g., see FIG. **6**, module **696**). Terms and conditions negotiation processing module **1404** may comprise a module that permits users to select from a database or file of standard terms and conditions (e.g., see FIG. **6**, module **697**). Contract formalization processing module **1406** may comprise a module that permits users to formalize or enter into a contract that has been negotiated. This may include the provision of digital signatures of one or more of the users to indicate their assent to the terms (e.g., see FIG. **6**, module **699**). Electronic funds transfer processing module **1408** may comprise a module that permits funds to be transferred electronically from one user to another. Thus, the entire contracting process, including drafting, negotiation and funds transfer can be conducted in the virtual environment. Other electronic commerce processing modules **1410** may comprise any other modules supporting the electronic commerce facilities of the system.

In one embodiment, electronic commerce processing module **988** may comprise a platform independent client-side application, such as a Java™ applet, that is provided to facilitate the processing tasks performed by one or more of modules **1402–1410**.

Engineering Analysis and Simulation (EAS) Processing

Engineering analysis and simulation processing module **946** permits users, such as EAS team members, to carry out various analyses and simulations and, as depicted in FIG. **15**, may comprise stress analysis processing module **1502**; system dynamics analysis processing module **1504**; rotordynamics analysis processing module **1506**; thermal analysis processing module **1508**; fluid dynamics analysis processing module **1510**; motion simulation module **1512**; mechanical event simulation module **1514**; assembly simulation module **1516**; materials analysis processing module **1518**; component interference analysis processing module **1520**; other analysis and simulation modules **1522**; machining process simulation module **1524**; casting simulation module **1526**; forging simulation module **1528**; sheet metal process simulation module **1530**; and other manufacturing simulation modules **1532**.

Stress analysis processing module **1502** may use finite element or equivalent numerical analysis techniques to determine the stress distribution throughout in a part design model. System dynamics analysis processing module **1504** may use numerical techniques to evaluate the dynamic behavior of a part design model, such as resonance points, acoustical properties and the like. Rotordynamics analysis processing module **1506** may use numerical techniques to evaluate behavior of rotating parts, such as to evaluate vibration performance at a design speed and through the transition from startup to the design speed. Thermal analysis processing module **1508** may use numerical techniques to evaluate a part design model at various temperatures in terms of stress magnitudes and deformation or weakening that may affect performance.

Fluid dynamics analysis processing module **1510** may use numerical techniques to evaluate performance of a part design in a fluid environment, and may include measurement of such parameters as pressure, temperature, and density distributions. Motion simulation module **1512** may use numerical simulation techniques to evaluate performance of a part design while in motion, such as to determine interference between components or with other objects in the operational environment, and to determine whether pressures and forces are excessive. Mechanical event simulation module **1514** may use numerical techniques to evaluate a part design model's behavior in response to mechanical events, such as crashes or collisions, and may provide predictions of the extent of deformation, dents and the like. Assembly simulation module **1516** may use numerical techniques to simulate the assembly process for a part design model to evaluate the producibility thereof.

Materials analysis processing module **1518** may use numerical techniques to evaluate a part design model in terms of the materials to be used. For example, certain entities may be designed to be made of Kevlar™, and this analysis may be used to evaluate behavior based on the properties of said material (e.g., see FIG. **7**, module **331**). Component interface analysis processing module **1520** may use numerical techniques to evaluate the interface between parts of a part design model, or between the part design model and external items. Other analysis and simulation modules **1522** refers to any other modules used to evaluate a part design model.

Modules **1524–1532** are especially relevant to fabricators evaluating producibility of a part design model. Machining process simulation module **1524** may be used to evaluate whether a part design model (or portion thereof) may be manufactured using particular machines. For example, the dimensions of the part design model may be considered to determine which machines may be used and what material stock may be used. Casting simulation module **1526** may be used to determine whether casting processes may be used to produce a part design model. Forging simulation module **1528** may be used to determine whether forging processes may be used to produce a part design model. Sheet metal process simulation module **1530** may be used to evaluate whether the part design model may be made of sheet stock. Other manufacturing simulation modules **1532** refers to any

other manufacturing simulations that may be offered by the NICECAD system.

For purposes of clarity, it should be noted that there may be correspondence between the analytical tools provided by modules **1502–1532**, especially those pertaining to the production engineering discipline, and the producibility logic supported by the GMR system (e.g, module **2650**, FIG. **21**). In general, modules **1502–1532** represent software which is used by design team members during the design and development phase to evaluate a proposed design. Typically, the current baseline part design model (or parts thereof) is the input, and the output is some measure of performance or compliance with applicable specifications. On the other hand, the producibility analysis performed by module **2650** (FIG. **21**) represents analysis performed by the server for the purposes of generating the second query set (to be discussed below) for purposes of searching the GMR database for qualified fabricators. Therefore, while the results of the producibility analysis may be made available, and while the substantive analysis may involve some of the same computations, the results are used for different applications in the two contexts (NICECAD and GMR searching).

Regarding implementation of the aforementioned modules, in one embodiment the modules may comprise executable files or code that may be accessed by the user for downloading and execution (e.g., the NICECAD system may present a browser page to the user, who could then select from a list of analysis and simulation options). In this embodiment, the "number crunching" associated with the analysis or simulation is performed by the user station system. In another embodiment, the aforementioned modules are made available to users through a platform independent application, such as a Java™ applet, whereby the user may assign the inputs for the analysis (such as the part design model and various analysis or simulation parameters) and the "number crunching" is substantially performed by NICECAD server system **200**. In this fashion, the processing demands on the user system are minimized, and little or no specialized hardware or software is required.

It should also be noted that EAS processing module **946** (or parts thereof) may be integrated or coordinated with CAD processing module **932** (or parts thereof). For example, an analysis that requires 2D sections of a part design model may include the execution of a module such as part design model visualization and manipulation processing module **1210** (FIG. **12**). Access to EAS processing modules may be constrained by access permissions assigned by an approval authority (e.g., see FIG. **11**, module **1116**). Check-out records may be stored whenever a team member accesses one of the EAS processing modules (e.g., see FIG. **11**, module **1104**).

Overview of a Method for a Concurrent Engineering Project using a Virtual Collaborative Environment

FIGS. **16A** and **16B** provide an overview of a method for undertaking a concurrent engineering project in a virtual collaborative environment, such as that provided in the system previously disclosed.

Referring to FIG. **16**, a prime contractor first assembles a series of teams, as in step **1602**, and the prime contractor establishes a project on the virtual system, as in step **1604**, which may rely on a software component such as PDM electronic document processing module **1102**. This may require the creation of contracts between the prime contractor and suppliers (FIG. **4**, module **415**), the prime contractor and the NICECAD system, and supplier(s) and the NICECAD system (FIG. **4**, module **410**). NICECAD system server **200** may set up accounts for suppliers and the prime contractor (FIG. **5**, module **320**) and basic project data in the database (FIG. **8**, PDM data **335**). User ID's and passwords may be established for the prime contractor and suppliers (FIG. **10**, module **1002**).

The design team may create a preliminary design, as in step **1606**, which could be done with the assistance of a software component such as part design model creation and editing processing module **1202** of FIG. **12** and baseline part design model management module **1106** of FIG. **11**. The preliminary baseline part design model(s) (for simplicity, we refer to the model for a part in the singular as the part design model, although those of ordinary skill can appreciate that a design may comprise a plurality of part design models) is stored by the NICECAD server system, as in step **1608**, which could be stored in stored baseline part design model data module **865** of FIG. **8**. The design team may also create preliminary PDM documents, such as a system specification, performance specification, project schedule or other generally non-graphical project documents, as in step **1610**. This step may be carried out using a software component such as PDM electronic document processing module **1102** of FIG. **11**. The preliminary PDM documents are stored by NICECAD server system **200**, as in step **1612**, which may be in a module such as stored product data management system electronic document data module **840** of FIG. **8**.

The prime contractor may assign design and analysis access permissions to teams or individual team members, as in step **1614** (e.g., see FIG. **8**, module **860**; FIG. **11**, module **1116**). The prime contractor may then activate the preliminary PDM system, as in step **1616** (e.g., see FIG. **11**, module **1102**). Those of skill in the art understand that product data management is a rigorous process that may involve strict procedures and significant documentation. In the early phases of the development process (such as during the creation of the initiatial design), the design may change significantly as different approaches are considered and traded off. Consequently, a less rigorous preliminary PDM system may be used during the initial stages (e.g., see from "initial idea" to "Design 1," FIG. **1**) so that the PDM requirements do not overburden the project at the early stage.

Based on authorizations assigned by the prime contractor, teams perform analysis and simulation to evaluate the preliminary baseline part design model according to various disciplines, as in step **1618** (e.g., see FIG. **9**, module **946**; FIG. **15**). Results may be stored in the NICECAD system (e.g., see FIG. **8**, module **885**; FIG. **11**, module **1114**). The teams may then discuss proposed modifications to the preliminary baseline part design model based on such analyses, as in step **1620**. This may be performed in the virtual NICECAD environment using quasi-real time video, audio and graphics (see FIG. **13**; FIG. **12**, module **1210**). Records of these sessions may be stored (e.g., see FIG. **8**, module **890**). Based on such discussions, the design team may make changes to the preliminary baseline part design model by modifying a working copy part design model, as in step **1622**. "Working copy" generally indicates that the model is a working copy, and does not necessarily represent an approved baseline. The working copy part design model, as modified, may then be stored by the NICECAD system, as in step **1624**, to a working copy storage such as that of FIG. **8**, module **892**.

The prime contractor (or other approval authority) may approve proposed modifications to the preliminary baseline part design model, as in step **1626**, based on the recommendations made by the design team and analysis team(s). This step could involve the creation of a digital approval signa-

ture (e.g., see FIG. 8, module 870; FIG. 11, module 1112), although in one embodiment no such formality is required while in the preliminary PDM phase. The prime contractor may then store the new preliminary baseline part design model, as in step 1628. In one embodiment, this new model may be stored in module 865 of FIG. 8 and a revision history in module 840 of FIG. 8 would be updated. Preliminary PDM documents may be updated as required for the new design, as in step 1630.

If the preliminary baseline part design model is still not sufficiently mature (decision block 1632, "No"), the prime contractor may request that additional analyses be performed, in which case the method returns to step 1618. If the preliminary baseline part design model is considered sufficiently mature (decision block 1632, "Yes"), the prime contractor may commence the formal PDM process, as in step 1634. Those of ordinary skill can understand that the point of "sufficient maturity" of a design is a determination that is based on the judgment and experience of the prime contractor project manager(s).

The methodology of the project development effort during formal PDM is similar to that of preliminary PDM, only the PDM management is more rigorous. Therefore, steps 1636–1648 are performed like steps 1618–1630. During formal PDM, however, the documentation requirements and data management are enhanced. For example, as previously discussed, version numbers may be assigned to each baseline part design model and to each working copy part design model. A revision history of the baseline part design model may be maintained. Strict check-in and check-out procedures may be enforced. Digital signatures may be required for any change to the baseline part design model and associated PDM documents (specifications and the like).

After each round of analysis and simulation, the prime contractor determines whether the baseline part design model is sufficiently mature to begin fabricating prototypes. If not (decision block 1650, "No"), the method returns to step 1636 for additional analysis. If the baseline part design model is considered mature (decision block 1650, "Yes"), the prime contractor may approve the final part design model and PDM documents, as in step 1652. In one embodiment, this may entail digital signatures approving the final part design model and PDM documents (e.g., see FIG. 8, module 870; FIG. 11, module 1112).

At this point, contracts between the prime contractor and fabricators may be entered into, as in step 1654, for the production of prototypes. The fabrication team may then produce prototypes in quantity based on the final part design model, as in step 1656.

A Method Using a Virtual Collaborative Environment to Create an Initial Design

FIG. 17 depicts a method for creating an initial design using the collaborative networked environment. A designer team member (hereinafter abbreviated as "designer") logs on to the system, as in step 1702. The system authenticates the user, as in step 1704 (e.g., see FIG. 10, module 1002). The designer selects NICECAD CAD software for launch, as in step 1706. In one embodiment, this could be carried out by part design model creation and editing processing module 1202 of FIG. 12, which may be coupled to modules 1204-1208 for parts, drawing symbols and standards/specifications to be used in creating the part design model. In one embodiment, the NICECAD CAD software may include a platform independent application, such as a Java™ applet, that provides for client/server interactive CAD processing. In another embodiment, the NICECAD CAD software may include server-provided semi- machine-dependent applications executable with standard operating systems.

The designer creates graphic entities that comprise a part design model using CAD tools and commands of the NICECAD CAD software, as in step 1708. CAD packages generally comprise a series of software tools and commands, as well known to those of skill in the art. The designer may use a set of attribute names to identify the entities, as in step 1710. The designer associates attributes and attribute values to selected entities, as in step 1712. The designer associates components from standard and custom parts libraries as needed (it may not be necessary), as in step 1714.

Steps 1710–1714 may be accomplished using data from certain modules, such as stored standard attribute and attribute values data module 845 of FIG. 8 for attributes; manufacturing standards and specifications data module 850 of FIG. 8 for standards/specifications; stored standard drawing symbols data module 880 of FIG. 8 for drawing symbols; and standard and custom parts library data module 875 of FIG. 8 for prebuilt parts. This may be accomplished using various processing modules, such as attributes and attribute values management module 1110 of FIG. 11; standard and custom parts library management module 1204 of FIG. 12; standard drawing symbols management module 1206 of FIG. 12; and manufacturing standards and specifications management module 1208 of FIG. 12.

The designer creates and edits PDM documents (preliminary or formal) to properly track the new part design model, as in step 1716. Processing modules such as PDM electronic document processing module 1102 of FIG. 11 and data modules such as stored product data management system electronic data module 840 of FIG. 8 may be used to carry out this step. In one embodiment, this step may comprise the provision of a platform independent GUI interface and/or application, such as browser pages and/or a Java™ applet, that permits the designer to select various options, such as whether to create a system specification or performance specification. It may allow the designer to decide options for the revision history (FIG. 8, module 840) and whether version numbers should be assigned to all baseline part design models (e.g., see FIG. 8, module 865) and/or working copy part design models (e.g., see FIG. 8, module 892).

The system then stores the new part design model, attributes, attribute values, and PDM documents to the system, as in step 1718. Processing modules from FIG. 11 and data modules from FIG. 8 may be employed.

A Method Using a Virtual Collaborative Environment for Engineering Analysis and Simulation

FIGS. 18A and 18B illustrate a method for using the system to perform engineering analysis and simulation using the virtual environment. Before disclosing the method, it should be appreciated that the terms "preprocessing," "analysis or simulation," and "postprocessing" are descriptive terms that represent a functional allocation of the analysis/simulation process. Those terms are used to explain the process and are not meant to imply that the EAS analysis process must be segregated in a particular fashion. As those of skill in the art can appreciate, "preprocessing," "analysis or simulation," and "postprocessing" may be part of a single software "run" or may pertain to several discrete acts. For example, in the case of stress analysis, there may be preprocessing by representing a part design model as a mesh or network of points ("finite elements"). Then there may be the finite element analysis that computes the stress at each point. Then there may be the postprocessing that converts the numerical results into a presentation (e.g., a graph). Those of ordinary skill should appreciate that such acts may be performed as three separate sub-analyses (e.g., a separate

module performing each and having its own set of results or ouput) or as part of a single analysis (e.g., a single module or software component performing the entire analysis).

Returning to FIGS. **18**A & B, the team member logs on to the NICECAD system, as in step **1802**, and the NICECAD system authenticates the team member, as in step **1804**. The team member requests access to a part design model, as in step **1806**. As discussed previously, there may be a plurality of part design models associated with a given engineering development project, so that the team member may seek access to one or more of these.

NICECAD server system **200** provides access to the part design model and PDM data consistent with the authorization for that team member, as in step **1808**. As previously discussed, a module such as design and access permissions management module **1116** of FIG. **11** may permit a prime contractor to assign access permissions for the part design model, PDM documents, and EAS processing modules. This data may be stored in a module such as stored design and analysis access permission data module **860** of FIG. **8**. Using modules such as these, NICECAD server system **200** could permit (or deny) access to the part design model and PDM data, according to step **1808**.

If the EAS to be conducted relies on external programs or processing modules (or even so-called pen-and-paper analysis), the system skips to step **1826** (decision block **1810**, "No"). If the EAS relies on NICECAD hosted or provided EAS modules, the system proceeds to step **1812** (decision block **1810**, "Yes"). In the latter case, EAS software for the analysis may be provided by the NICECAD system.

In step **1812**, the team member launches an EAS preprocessing software component consistent with authorization. As noted above, the prime contractor may assign access permission data for the EAS modules, so the team member must be authorized to access that EAS software component. In one embodiment, launching the preprocessing software component comprises running a software program that is downloaded from the server and run on the team member's user system. In another embodiment, launching comprises running a platform independent application, such as a Java™ applet, that is interactive with NICECAD server **200** so that the bulk of the processing is performed by NICECAD server **200**.

In step **1814**, the team member performs preprocessing procedures for the analysis or simulation. In step **1816**, the team member launches the analysis or simulation software component consistent with authorization (see step **1812** for several embodiments). In step **1818**, the team member performs the analysis or simulation, which may require a series of computations. As noted with respect to step **1816**, the step may comprise running a platform independent application, such as a Java™ applet, so that the performance of the analysis or simulation may actually be a shared task between the server and the team member's user system.

In step **1820**, the team member launches the analysis or simulation postprocessing software component consistent with authorization, and in step **1822**, the team member performs the postprocessing procedures. In step **1824**, the team member posts results to NICECAD server system **200** and updates appropriate PDM documents. A processing module such as EAS results management module **1114** of FIG. **11** and data module such as stored EAS results data module **885** of FIG. **8** may be used to carry out this step. While step **1824** states that the team member posts the results, in one embodiment the NICECAD system may provide that all EAS results be automatically stored.

Upon completion of the analysis or simulation, step **1834** provides that team members discuss effects of the EAS results on the part design model. Generally, this discussion may include one or more members from the design team and the particular EAS team, although this may vary based on circumstances. As discussed with respect to FIG. **13**, such discussions may take place over the networked virtual environment using quasi-real time audio and/or video and/or graphics. Based on said discussions, the prime contractor makes a decision on whether a design change to the part design model should be effected, as in step **1836**.

If the analysis or simulation is to rely on external software (decision block **1810**, "No"), the team member may request the download of the part design model, as in step **1826**. In step **1828**, NICECAD server system **200** downloads the part design model. In one embodiment, steps **1826**–**1828** may include giving the team member the option to select a file format (e.g., IGES, STEP, etc.) for the downloaded part design model. In one embodiment, this step may be facilitated by providing browser pages and/or a Java™ applet or the like. In step **1830**, the team member performs the external analysis or simulation. In step **1832**, the team member uploads the results to the NICECAD system and updates appropriate PDM documents. Steps **1834** and **1836** are performed as previously discussed.

A Method Using a Virtual Collaborative Environment for Quasi-Real Time Interaction

FIGS. **19**A & B illustrate a method for team members to engage in an interactive communications session using the system. As can be readily appreciated, the multimedia communications capability has ready applications in all three aspects of the present system: the development of the design, the search for qualified fabricators, and the negotiation and bidding process. The detailed description below describes the communications session in the context of design development, but this is exemplary only, and the multimedia communications capability finds ready application in the other aspects of the invention.

In step **1902**, participating team members log on to the NICECAD system, and in step **1904**, the NICECAD system authenticates the team members. In step **1906**, the team members request a secure interactive session using audio and/or video. In one embodiment, the system may provide additional levels of security for such a session, such as providing encryption with greater key lengths. Continuing with step **1908**, NICECAD server system **200** launches the interactive video and/or audio software component(s) (e.g., see FIG. **13**, modules **1302**–**1304**). Steps **1906**–**1908** may be carried out through the use of a platform independent application, such as a Java™ applet.

In step **1909**, the session mediator establishes the new session and coordinates activity among the participating team members. The "session mediator" generally refers to the team member who has responsibility for the interactive session and acts as the "virtual chairperson." In one embodiment, the session mediator establishes the new session by providing the identity of the participating team members and a description of the session (e.g., "NICECAD session of Jun. 1, 2010, to discuss proposed design changes as a result of stress analysis") to NICECAD server **200**.

If the interactive session is not going to require a part design model (decision block **1910**, "No"), the method will skip to step **1916**, discussed below.

If the interactive session is going to need a part design model (decision block **1910**, "Yes"), the method proceeds to step **1912**. In step **1912**, the NICECAD server system **200** launches the interactive graphics software component (e.g.,

see FIG. 13, module 1306). According to step 1914, NICECAD system server then commences network transport and management of the audio and/or video and graphics data (e.g., see FIG. 13, modules 1302–1306). NICECAD server 200 retrieves the part design model requested by the session mediator, as in step 1922. The part design model is presented to the participating team members (e.g., see FIG. 12, module 1210; FIG. 13, module 1306), as in step 1924, and the team members interact using audio and/or video to discuss design issues, as in step 1926.

In step 1928, team members may interact by processing or manipulating the part design model using interactive software supplied by NICECAD server system 200 (e.g., see FIG. 12, modules 1210 and 1202). In one embodiment, NICECAD server system 200 permits only one team member to control the part design model at a time, and after each change, the results are presented to all participating team members.

Step 1928 may be further illustrated by subdividing it into steps 1930–1934. In step 1930, a team member rotates or translates or otherwise manipulates the part design model to focus particular design issues. In step 1932, a team member applies comments or "mark ups" to the part design model to highlight particular aspects of the part design model. In step 1934, the team member applies the virtual "cutting tool" to generate 2D sections of the part design model. As previously described, module 1210 may support "fly through" functionality, so that users can also simultaneously view internal regions of the part design.

In step 1936, a decision is made regarding the design issues, and the part design model and PDM documents are updated as required. In one embodiment, only the prime contractor has approval authority for design changes. In another embodiment, the NICECAD system may permit the prime contractor to delegate this authority (such as for a particular interactive session) at its discretion. In step 1938, the interactive session is saved to the NICECAD system (e.g., see FIG. 8, module 890).

Returning to the "No" branch of decision block 1910, the interactive session may not require the part design model, in which case audio and/or video may be used. In step 1916, the NICECAD server commences the network transport and management of the audio and/or video. In step 1918, the team members interact to discuss issues of interest, such as design issues, contract issues and so on. The method continues with steps 1936 and 1938 as previously discussed.

A Second Aspect of the Invention: A System and Methods for Providing a Searchable Database of Registered Fabricators for Producing a Part Design (GMR)

A second aspect of the invention is directed to a system and methods for providing a searchable registry of fabricators which can be searched by a designer or prime contractor seeking to manufacture a product represented by a part design model. As can be readily appreciated, the design (represented by a part design model and specifications) may have been developed using the previously described system and methods (the NICECAD aspect). In such a scenario, much or all of the necessary design information resides on the system (e.g., the three dimensional part design model may be stored in module 865 and the specification documents may be stored in module 840, both of FIG. 8). However, the overall system provides maximum flexibility insofar that the second aspect of the invention has application even where the design is not already stored on the system. Therefore, the second aspect of the invention, the searchable GMR registry, permits a designer or prime contractor to upload and, if necessary, convert an independently-developed part design model in order to locate qualified fabricators.

Overview of the Searchable GMR

FIG. 20 depicts an overview of the second aspect of the invention, which is generally referred to herein as the searchable Global Manufacturer's Registry (GMR). In the illustrated embodiment, there is GMR server system 1000, databases 1100, designer user systems 1200, fabricator user systems 1300, network interface 1600, backup server hardware and software 1400 and backup data storage 1500.

As previously noted, the illustrated architecture is exemplary insofar that GMR Sever System 1000 and NICECAD Server System 200 could easily be hosted on the same server system. For clarity, FIGS. 20–25 depict and the following description focuses on the second aspect of the invention as an independent system. However, in the preferred embodiment both aspects are implemented as an integrated system. In that case, network interface 1600 (FIG. 20) and network interface 260 (FIG. 2) are the same. Likewise, database 1100 (FIG. 20) and database 210 (FIG. 2) may comprise elements of an integrated database. Backup server hardware and software 1400 and backup data storage 1500 (FIG. 20) and backup server hardware and software 240 and backup data storage 250 (FIG. 2) could be integrated. And finally, it should be noted that the terms "fabricator" user systems 1300 and "designer" user systems 1200 (both of FIG. 20) and "prime contractor" user systems 220 and "supplier" user systems 230 (both of FIG. 2) are broadly descriptive. A designer searching the GMR database may be the prime contractor, and an outsourced fabricator is considered to be a supplier.

Network interface 1600 may comprise any network that allows communication amongst the components, and may encompass existing or future network technologies, as previously discussed in connection with network 260 (FIG. 2). In one embodiment, use of the Internet as network 1600 is beneficial since it may maximize the universe of designers and fabricators who may participate in the GMR system.

Designer user systems 1200 may comprise any system capable of interfacing with network 1600. Designer user systems 1200 may comprise "standard" computer systems that do not require specialized hardware or software to use the GMR system. Designer user systems 1200 may comprise personal computers or any like computer systems described in connection with prime contractor user systems 220 (FIG. 2). In the preferred embodiment, designer user system 120 comprises a personal computer or workstation running a standard operating system such as Windows NT, and using a standard browser such as Microsoft Internet Explorer™ 5.0 capable of interpreting HTML 4.0, XML, VRML, and running Java™ applets, so as to support a substantially platform-independent interface with GMR Server System 1000. As discussed previously in connection with the NICECAD aspect of the integrated system, user systems such as designer user systems 1200 may comprise computer systems using a custom browser tailored to the present application. As before, in other embodiments, such user systems may be executing server-provided semi- machine-dependent applications, particularly for graphics-intensive applications, which might be coded in a language such as Open GL™. Even where such applications are executed by user systems, a platform independent network interface can be maintained.

Similarly, fabricator user systems 1300 may comprise any system that may interact with the central server system over network 1600.

Backup data storage 1500 comprises a system for backing up the data stored by the interactive EDM system, and can

comprise the storage technologies and/or architectures discussed previously in connection with module **250** of FIG. **2**.

Backup server hardware and software **1400** comprises one or more backup servers, including server modules, to reliably support the second aspect of the invention, and may comprise backup means discussed in connection with module **240** of FIG. **2**.

Databases **1100** depict the storage that may be employed to store data maintained by the GMR system, including storage technologies discussed in connection with database **210** of FIG. **2**. Databases **1100** may store the fabricator specific information stored in Global Manufacturers Registry data module (e.g., see FIG. **22**, block **3000**), as well as designer account data, fabricator account data, and other data. In one embodiment, databases **1100** resides locally with GMR server system **1000**, although future network technologies supporting greater bandwidth may support remote location.

GMR server system **1000** comprises a server system supporting the interactive GMR system. GMR server system **1000** interfaces with designer user systems **1200** and fabricator user systems **1300** through network **1600**. GMR server system **1000** may include the hardware and software to interface with designer and fabricator user systems on a substantially platform independent basis so that the designer and fabricator user systems do not require their own specialized hardware or software. Generally, GMR server system **1000** includes system administration, database manipulation, and network-related operations software modules. GMR server system **1000** may also include the specialized software, some server side and some client side interactive, for supporting upload, conversion, processing and analysis of part design models, such as two-dimensional models or three-dimensional solid models.

GMR server system **1000** may also include the hardware and software for multimedia operations such as voice, video and graphics, that may be transmitted over the network for presentation to user systems with standard client-side multimedia support. In the preferred embodiment, where the first and second aspects of the invention are part of an integrated system, this multimedia support is provided by multimedia communications processing module **978** (FIG. **13**) and multimedia session records are stored at module **890** (FIG. **8**). The CAD capability for permitting multimedia sessions involving the viewing and manipulation of a part design model are provided by CAD processing module **932**.

In one embodiment, GMR server system **1000** may be publicly accessible as a web site, but may provide multiple levels of security (e.g., multiple "firewalls") to ensure that proprietary designer and fabricator data may be protected. In addition to firewalls, strong (e.g., RSA 1024 bit) encryption may be provided to protect proprietary data. GMR server system **1000** may be accessed only by users with a user ID and password. In one embodiment, GMR server system **1000** may comprise a "back end" processing server running UNIX for the model processing and database operations, and a "front end" web server running Windows NT™ and Microsoft's Transaction Server™ for network-related operations. Of course, where the first and second aspects of the invention are integrated, then such functions are performed by system administration processing module **902** (FIG. **9**).

The GMR Server System

FIG. **21** illustrates some of the various software components or modules of GMR server system **1000**. GMR server system **1000** may comprise user logon authentication module **2050**; designer service authorization module **2100**; contract processing module **2150**; report generation module **2200**; billing information processing and report generation module **2250**; fabricator update processing module **2300**; search initiation module **2320**; general GMR search query processing module **2350**; interactive designer preference module **2400**; interactive designer part design model processing module **2450**; part design model file parsing module **2500**; interactive designer attribute assignment module **2550**; user preference first query set processing module **2600**; producibility analysis second query processing module **2650**; query results reconciliation processing module **2700**; GMR designer account processing module **2750**; GMR fabricator account processing module **2800**; other server side applications **2850**; quality assurance processing module **2905** and other client side interactive applications **2900**. As previously noted, where the NICECAD and GMR aspects of the invention are integrated, some of these modules may be subsumed in similar modules described in connection with the first aspect of the invention.

User logon authentication module **2050** may comprise a security module for limiting access to the interactive GMR system. Designer service authorization module **2100** may comprise a module that sends designer authorization information including, for example, information to tell the designer requesting a service, such as a search request, that the request may result in a billing action and that proceeding further constitutes final authorization. Contracts processing module **2150** may comprise a module that ensures that before a fabricator registers with the system, and before a designer uses the system, contracts specifying the responsibilities of each party regarding data protection, maintenance and system usage are formalized. Report generation module **2200** may comprise a module that formats and sends reports requested by users that, in one embodiment, may be sent as browser pages or links to downloadable files. Billing information processing and report generation module **2250** may be a module that processes and stores billing information whenever a billable transaction occurs. Billing information processing and report generation module **2250** generally automatically processes transactions as they occur and, in one embodiment, may also process billing summary reports upon user request.

Fabricator update processing module **2300** may be a module that allows fabricators to load their fabricator specific data into the database, as well as update their data when their background or capabilities change (e.g., the manufacturing facility relocates or a new machine is purchased). One of the advantages provided by the GMR capability of the invention is that it not only provides a searchable database for designers/prime contractors, but it also can function as a capability management system for fabricators. The fabricators can input, update and otherwise maintain an inventory of their capabilities and experience that can be used for capability management and marketing independent of its application for GMR searches. This incidental benefit to the GMR system provides an incentive for fabricators to register in the first instance. It also means that the burden of maintaining the fabricator capability data does not rest on the GMR system provider.

Search initiation module **2320** may be a module that permits a user to initiate a search session. In one embodiment, search initiation module **2320** may present to the user's browser a "search page" from which particular searches can be launched.

General GMR search query processing module **2350** may be a module that permits a user to conduct a general search of the GMR database for data that may be generally accessible (or nonproprietary). In one embodiment, general GMR

search query processing module **2350** supports searches by both designers and fabricators who wish to access general information regarding registered fabricators. Generally, a so-called general search involves search criteria that are not directly related to a design. For example, a general search may query such information as a fabricator's location, shop certifications (e.g., ISO 9000/9001/9002), operator certifications (e.g., are CNC milling machine operators factory trained?), and like information.

Interactive designer preference module **2400** may comprise a module that permits a designer to input so-called "preference" data in order to search the GMR database based on a limited GMR search. For a limited GMR search, the designer is not required to upload a part design model, but may select a series of criteria for qualifying fabricators (such as lead times, material, lot sizes, machinist/craftsman certifications, etc.) that are considered global. Unlike a general search, the limited search may result in the search of limited access (or proprietary) fabricator data (e.g., see FIG. **22**, blocks **3200**–**3400**) and, consequently, this option may be limited to designers. In one embodiment, the designer may select a set of preferences from a browser interface with GMR server system **1000** that, following confirmation, may be temporarily stored and tagged with owner ID, session ID and time stamp (see FIG. **22**, stored designer preference data module **3550**). These preference inputs may then be processed to generate a first query set. It should be noted that while a preference query may search proprietary data to identify qualifying fabricators, the underlying proprietary data is generally not provided to a searching designer. The fabricator "owns" the proprietary data maintained on the GMR system, and generally only the owner can view and edit said proprietary data.

Interactive designer part design model processing module **2450** may comprise a module that permits a designer to upload a file representing a product or part to be fabricated (the term part design model used herein generally refers to such a file). According to one embodiment, the part design model may comprise a three-dimensional (3D) solid model. In another embodiment, the part design model may comprise a two-dimensional (2D) model. In one embodiment, interactive designer part design model processing module **2450** sends a page to the designer's browser that provides for the launch of a client-side Java™ applet that may be executed to permit the following operations: browse local memory for the file, select the file format, and upload the file to GMR server system **1000**. In one embodiment, interactive designer part design model processing module **2450** may convert the uploaded part design model to a neutral or common format, such as an AP214 STEP file, so that stored models share the same file format. This furthers the data neutrality attribute of the invention previously discussed. In one embodiment, interactive designer part design model processing module **2450** may also provide utilities for the designer, such as for the conversion of solid model files among different file formats, such as between IGES, STEP AP203 and AP214 model formats and other CAD-vendor specific formats, known to those of skill in the art.

Part design model file parsing module **2500** may comprise a module that identifies graphical entities (e.g., geometric and/or topological features) contained in an uploaded part design model file and converts them to a neutral format before storage: In one embodiment, each entity may be tagged with an owner ID, session ID and time stamp before storage in an object database (e.g., see FIG. **22**, module **3600**). If the uploaded file contains non-graphical data such as geometric dimensioning and tolerancing information (known to those of skill in the art as GDT data), these data items may be extracted, converted to a neutral format, and tagged before storage in the object database. Part design model file parsing module **2500** may also perform a check to ensure that no duplicate entities exist.

Interactive designer attribute assignment module **2550** may comprise a module that permits a designer to assign attributes to entities of the stored part design model. A part design model may comprise a series of geometrical and topological entities, each of which may be assigned its own attributes (such as surface finish, material, direction vector, etc.).

In one embodiment, interactive designer attribute assignment module **2550** may provide for the launch of a Java™ applet or like miniapplication at designer user system **1200** for attribute assignment. A textual list and graphical presentation of the model entities may be sent by GMR server system **1000** to designer user system **1200** for presentation. A list of standard attributes (e.g., see FIG. **22**, stored standard attributes data module **3500**) may also be sent by GMR server system **1000** to designer user system **1200** to facilitate the process. The list may include such items as feature name (so the designer may name the entity), tolerance type (e.g., flatness, linear, angular, runout, etc.), tolerance range, surface finish value, and fabrication standard (e.g., MILSPEC, ISO, ANSI, ASME, and others known to those of skill in the art). For example, the designer may also input direction vectors for each entity that define the outside surface of the model. By keyboard entry or "point and click" use of a mouse (or similar pointing device), the designer may be enabled to select entities and assign attributes and attribute values. Upon completion of attribute assignment, the assigned information may be associated with the entities stored by GMR server system **1000** in the database. It should be noted that, in the case of a "complete" solid model (e.g., a STEP AP214 file), sufficient information may exist for an acceptable producibility analysis to be run without further inputs, in which case attribute assignment may be avoided.

User preference first query processing module **2600** may comprise a module that prepares a first query as a result of the designer's preference inputs (see interactive designer preference module **2400** above) which may be run against the database to identify a first results list of qualified fabricators. In the case of a limited search, this may be the end of the designer session. In a limited search, appropriate results (e.g., the identity of all qualified fabricators) may be stored in the designer's account data and appropriate results (e.g., referral results providing basic information about the designer) may be stored in each qualified fabricator's account data. Notice may be then given to the designer and qualified fabricators that results have been posted. For a fill search, on the other hand, the first results list may be combined with the second results list before posting, as described in detail below.

An non-exhaustive list of the types of information in a preference query includes: whether the fabricator is ISO9001 certified; whether the vendor has specified trade association affiliations; whether the vendor is a minority-owned business or meets other requirements or standards under the federal acquisition statutes and regulations; whether the vendor meets the specified GMR quality assurance rating (discussed further below); whether the vendor is within **1000** miles of the designer's location; whether the vendor can support specified transportation modes, such as air, rail, and truck; whether the vendor can operate in a so-called "just-in-time" inventory environment; whether the vendor has the ability to process Catia™ CAD/CAM design

data; how many MILSPEC contracts has the vendor completed successfully; and has the vendor breached past contracts or otherwise fail to fully meet past contractual obligations.

For a full search (which may include the upload of a part design model, producibility analysis and a second query, among other steps), producibility analysis second query processing module **2650** performs a producibility analysis of the uploaded, converted and attribute-assigned part design model in order to prepare the second query set for a second results list of qualified fabricators.

One Example of Producibility Analysis Logic

As one of ordinary skill can appreciate, various steps may be performed for the producibility analysis. According to one embodiment, the steps performed ensure that the material from which a part is to be made is compatible with any preferred processes selected by the designer. If no preferred processes are specified, all processes for which the material may be acceptable are identified for inclusion in the second query set.

Each surface entity may be checked for a direction vector defined by the designer; if none exists, the system attempts to define a direction vector for all surfaces in order to permit an interference check. Each surface may be then projected along its direction vector in order to check for interference. If interference exists, then the conflicting surface entities are noted and the designer may be given appropriate notice (e.g., by e-mail) so that the conflict may be resolved (e.g., by modifying the attribute assignment). The results of the interference check may be used to ensure that designer-selected processes are appropriate for the part, particularly to ascertain any machine tool/die interference in the case of machining, forging, sheet metal and other such processes. For example, a forged part must have surfaces shaped in such a way as to permit a die press (e.g., drop forge) to fall along a straight path when the blank is struck. For example, a complex part may not be machinable, but may be cast and then finished with slurry processing and machining processes. If no processes are specified by the designer, all appropriate fabrication processes identified by this analysis may be included in the second query set.

The system may perform a volume approximation of the part represented by a 3D part design model in order to assess producibility using various processes, such as casting, where volume may be an important consideration.

The part design model may also be evaluated for the existence of passages or voids (e.g., closed channels, holes and other internal features that may require material removal). The size and topology of these passages or voids may be evaluated for producibility using various processes, such as machining, where passage topology may be an important consideration.

The part design model may also be evaluated based on an envelope enclosing the part represented by the model. The dimensions of the envelope may be used to assess which machines may be used to manufacture the part and also what material stock may be used. For example, a cylindrical part will likely be made using a lathe and may require rod stock of a certain size.

The producibility analysis steps may also estimate the part design model thickness to determine if the part may be made of sheet stock, and to what extent specific features or entities may be produced using sheet metal processes.

Each entity assigned a tolerance may be evaluated for whether it may be achieved for the selected processes. If no process is specified, all processes capable of achieving the desired tolerance may be included in the second query set.

The part design model may be evaluated for topological symmetry to estimate the suitability of symmetrical manufacturing processes such as spinning or lathing. Acceptable processes may then be included in the second query set.

For each surface finish specified by the designer, all finishing processes capable of achieving that finish may be identified and included in the second query set. Each specified surface finish, as well as each tolerance, may also be evaluated to ensure compatibility with the overall part design model topology.

As one of ordinary skill in the art can appreciate, there are a multitude of variations of the producibility analysis that do not depart from the spirit of the invention disclosed herein, which is to employ some automated logic to perform part-specific analysis in order to identify fabricators.

Producibility analysis second query processing module **2650** may be a module that uses the results of the producibility analysis to prepare a second query set. The second query set may be compared to databases **1100** to generate a second results list of qualified fabricators. In one embodiment, the raw producibility analysis results, including producibility analysis methodology, may be also be stored and made available to the designer.

When a full search (i.e., using the first query set based on preference data and the second query based on the part design model/producibility analysis) is conducted, multiple sets of qualified fabricators may be produced. A predetermined number, such as two sets, may be produced, for example, one set corresponding to each query. Query results reconciliation processing module **2700** reconciles the produced fabricators to render a single list of qualifying fabricators (e.g., for two sets of results, those appearing in both result lists). If the results cannot be reconciled (e.g., there are no qualified fabricators appearing in both lists), the results may be analyzed to identify the most likely cause of the reconciliation failure. This information may then be stored in the designer's results report.

GMR designer account processing module **2750** may comprise a module that maintains and updates all account data for each designer. In particular, this module may provide that records are stored for each designer session (a designer session may be one complete cycle of a search), including the search results, as well as all transactions (a designer transaction refers to any time the designer logs onto the system, such as to retrieve results or have a custom report prepared). In one embodiment, GMR designer account processing module **2750** may provide that a designer user ID, session ID, time stamp and descriptive information (including the search results) be stored following each completed session. In one embodiment, GMR designer account processing module **2750** provides that a designer user ID, session ID, and time stamp are stored following each transaction (e.g., if the designer **99** logs on to have the results from sessions **101** and **102** downloaded at 1:30 p.m., then all of this information may be recorded as a transaction).

GMR fabricator account processing module **2800** is similar to GMR designer account processing module **2750**, only the former may record account data for each registered fabricator. GMR fabricator account processing module **2800** may store records for each session (a session may be a general search by the fabricator, GMR input/update of fabricator specific data, or the referral results emanating from a designer search that identified this fabricator as qualified) and each fabricator transaction (a fabricator transaction refers to any time the fabricator logs onto the system, such as to retrieve referral results or have a report

generated), or may only store some sessions or transactions. In one embodiment, a fabricator user ID, session ID and time stamp may be recorded for each session and transaction. For example, when a designer search identifies a particular fabricator as qualified, the referral results may be posted in the fabricator's account by this module.

Quality assurance processing module **2905** comprises a module that provides the quality assurance rating capability of the system. Following the performance (or lack thereof) of a fabrication effort by a GMR-registered fabricator, the prime contractor is given an opportunity to provide feedback to the GMR system regarding performance in terms of timeliness, quality, and other relevant considerations. In one embodiment, GMR server system **1000** may be accessed by the prime contractor for browser pages or the like to provide a performance assessment. The quality assurance survey may include a series of questions which can be answered quantitatively (e.g., answered as 1–5, with "5" being "excellent" or "strongly agreed" and "1" being "poor" or "strongly disagree"). Such data can be compiled and statistically aggregated into a series, or one overall, quality assurance rating. The quality assurance browser forms may also include questions to be answered in a narrative manner, not necessarily to be compiled, but to be made available to subsequent users of the GMR system. Quality assurance processing module **2905** supports not only the collection and compilation of such data, but also the distribution of the data to system users.

Other server side applications **2850** refers to any other predominantly server side applications run by GMR server system **1000** to support the interactive EDM system. For example, this module may include software for system administration, database manipulation, and network-related operations. This module may include software to support multimedia applications such as the transport of voice, video and graphics over the network.

Other client side interactive applications **2900** refers to any other predominantly client side applications that may be supported by GMR server system **1000**. For example, if multimedia operations require specialized capabilities exceeding standard user system multimedia support, other client side interactive applications **2900** may provide those capabilities in the form of downloadable executable code. In one embodiment, such client-side capabilities are supported through the use of machine independent Java™ applets supported by standard browsers.

The Global Manufacturer's Registry (GMR) Database

Databases **1100** contains the data used by the GMR system. Internal analysis criteria data module **3450** represents any stored constants and other data used to implement the producibility logic (e.g., see FIG. **2**, module **2650**). Stored standard attributes data module **3500** represents attributes, and may also include attribute values, that may be made available to designer user system **1200** to select for attribute assignment (e.g., see FIG. **2**, module **2550**). In one embodiment, this data may be transmitted to an applet launched at designer user system **1200** so that the designer may readily assign attributes and values to selected graphical entities. In the integrated embodiment, module **3500** is integrated with module **845** of FIG. **8**. Stored designer preference data module **3550** may comprise stored preference data that may be input by the designer for the first query set (e.g., see FIG. **2**, module **2400**). Stored designer part design model data module **3600** may comprise the stored part design models, including graphical and non-graphical entities and any associated attributes. As noted above, in one embodiment, each entity may be appropriately tagged to facilitate data management.

Account data **3900** may comprise account data maintained by the system for designers and fabricators. In one embodiment, account data **3900** may comprise both session records (such as searches, including search results; referrals, including description of the referring designer; GMR updates; etc.), and transaction records (such as results retrieval, report printouts, etc.). Account data **3900** may be further logically divisible into designer account data module **3800** and fabricator account data module **3850**. Where the GMR and NICECAD aspects of the system are part of the integrated embodiment, account data **3900** may be combined with account data **320** of FIG. **3**.

Quality assurance data module **3605** comprises quality assurance data collected reflecting the performance of fabricators on the GMR system. The module may comprise both the underlying data submitted by designers/prime contractors using electronic survey forms or the like, as well as the aggregated GMR quality assurance rating that derives therefrom. While FIG. **22** depicts quality assurance data module separate from GMR data module **3000**, it should be appreciated the GMR quality assurance rating(s) are searchable by system users. For example, a preference query (to be discussed below) might include criteria regarding quality assurance rating, such as to identify only fabricators with a net rating of "3" or better.

Global Manufacturers Registry (GMR) data module **3000** comprises fabricator specific data stored for registered fabricators. As indicated by the arrows on FIG. **22**, the module may contain both general access (non-proprietary) data and limited access (proprietary) data. In one embodiment, limited access data may be directly accessible only by the owner (e.g., the fabricator that submitted the data). In one embodiment, the data owner may also give authorization for particular users to access the owner's proprietary data. In general, the interactive GMR system itself may issue queries that search both types of data to locate qualified fabricators in response to a designer search request. This may be the case even though the results presented to the designer contain no proprietary information.

As those of ordinary skill should appreciate, GMR data module **3000** may contain a multitude of different types of fabricator information that may be used for designer searches, so that the categories depicted below are illustrative only and non-exhaustive.

General information group data module **3250** may contain general information, such as location, certifications and affiliations, transportation modes, CAD capabilities, trade restrictions (import/export tariffs, technology transfer restrictions, etc.), electronic commerce capability, computer hardware/software capability, status as federally-recognized minority-owned, small business or other special categories, and other items not related to specific machinery or manufacturing capabilities.

Process machinery group data module **3200** may contain information pertaining to the machinery and processes used by the fabricator. Information such as machine models, stock types and sizes, tolerances associated with the machine, production rates, lead times, set-up costs and times, numerical control systems, and other information may be included. Since different machines may support different processes, the inputs may vary from machine type to machine type. For example, a forging process may have a different input set than a vertical mill. Information about specific process capabilities may be included, such as electro-chemical machining, core casting, investment casting and powdered metal processing. Information pertaining to outsourced processes may also be included.

Tool set group data module **3150** allows a fabricator to catalog standard and custom tools used by machines. In one embodiment, standard machine tool lists (listing such items as end mill cutters, drills, abrasives, etc.) may be provided to limit the amount of information required by the fabricator. In one embodiment, an interactive application in the form of a Java™ applet may be provided to fabricator user system **1300** so that custom tools may be built interactively from a component list (e.g., see FIG. **21**, module **2300**).

Personnel operator group data module **3100** may contain information about the machinists, craftsman and operators of machines and processes, including years of experience and any certifications. In one embodiment, privacy may be maintained by limiting input to non-personal data.

Materials group data module **3300** may comprise information regarding those materials the fabricator can work. For example, maximum and minimum stock sizes, stock types (e.g., billet, rod, bar, sheet, powder, etc.) and material types (e.g., steel, aluminum, titanium, etc.) may be specified. In one embodiment, the fabricator may select from a standard list of materials, including material stock types, provided to the fabricator.

Manufacturing standards data module **3350** comprises data regarding those standards to which a fabricator can manufacture a part (e.g., MILSPEC, ISO, ANSI, ASME, and others well known to those of skill in the art).

Inspection/QA data module **3400** comprises fabricator-specified data regarding inspection processes, equipment and applicable standards. In other words, the fabricator may specify what methods are used to verify that a part has been made to designer specification and under what standards inspections are conducted.

As should be appreciated by those of ordinary skill, a variety of database structures and database management programs may be used for databases **1100**. In one embodiment, an object-oriented database structure may be employed combining Oracle 8™ and ObjectStore™ software, well known to those of skill in the art.

A Method for a Search Session using the GMR System

According to an embodiment of the present invention, a method for a designer search session using the GMR system is described in FIG. **23**. Referring to FIG. **23**, the designer first logs onto the interactive system, as in step **4000**, by connecting designer user system **1200** to GMR server system **1000** through network **1600**. The system is generally preferred to be secure, so that GMR server system **1000** authenticates or verifies that the designer is an authorized user, as in step **4020**. GMR server system **1000** may carry out this function through a software component such as user logon authentication module **2050**, depicted in FIG. **21**.

Following authentication, the designer selects the scope of services that are requested for the search session. According to one embodiment, the designer may select a general search of non-proprietary data, as in step **4040**. The non-proprietary data may be data such as that stored in general information group data module **3250**, depicted in FIG. **22**. In one embodiment, the designer makes this selection from a search page presented on a browser at designer user system **1200**. This may be performed by a software component such as search initiation module **2320**, in FIG. **21**.

If the designer requests a general search, then the designer may select search criteria for the search of non-proprietary data, as in step **4300**. For example, GMR server system **1000** may transmit a list of options for each of the data items contained in general information group data module **3250**. According to this embodiment, the designer may select options such as desired location (e.g. East Coast or Pennsylvania fabricators), certifications/affiliations and CAD capabilities through the browser at designer user system **1200**.

Based on input criteria selected by the designer, GMR server system **1000** prepares a query set appropriate for the general search, as in step **4320**, which may then be run against general access fabricator data, as in step **4340**, to identify fabricators meeting the criteria set by the designer. Steps **4320** and **4340** may be carried out by a software component such as the general GMR search query processing module **2350** depicted in FIG. **21**. The results of the general search carried out by step **4340** may then be stored in the designer account data, as in step **4360**, and sent to the requesting designer, as in step **4380**.

In one embodiment, the general search results may be tagged with a user ID, session ID and time stamp before storage in the designer account, such as that depicted as designer account data module **3800** of FIG. **22**, so that the designer may later retrieve them when desired. Thus, step **4380** may encompass a subsidiary step of GMR server system **1000** issuing an e-mail to the designer giving notice that the results have been posted. Upon such notice, the designer may log onto the system to retrieve the results by selecting the appropriate session in the designer's account for output as a report, to be discussed later in connection with FIG. **24**.

Once the search has been conducted and the results posted, the designer account data may be updated to reflect completion of the task (including storage of the results), as in step **4280**, as well as the corresponding fabricator account data, which may be updated for those fabricators identified as a result of the search, as in step **4400**. Steps **4280** and **4400** may be carried out by software components such as those depicted as modules **2750** and **2800**, respectively, in FIG. **21**. Finally, any applicable billing information may be generated and sent to the designer, as in step **4420**, and to fabricators identified as a result of the search, as in step **4440**. These billing processing steps may be carried out by a software component such as module **2250** of FIG. **21**, and may be stored in designer account data module **3800** and fabricator account data module **3850** of FIG. **22**.

208 Returning to the "no" branch of decision block **4040**, the designer may desire a search of the entire fabricator database, including proprietary data. As previously mentioned, the preferred embodiment provides that a limited search searches proprietary data to identify qualified fabricators to a searching designer, but the underlying proprietary data is not revealed to the designer. In this case, the designer may select between a limited preference search or full search according to step **4060**. As discussed above in connection with the system of FIG. **21**, a limited preference search may result in a first query set, and first query results list, without necessitating a producibility analysis. A full search uses the part design model and the producibility analysis so that a second query set and second query results list may be produced. If the user desires a limited preference search, the method skips to step **4160**, to be discussed below.

Otherwise, the designer may upload a part design model to GMR server system **1000**, as in step **4080**. Interactive designer part design model processing module **2450** may perform step **4080** using a platform independent Java™ applet. Where the NICECAD and GMR aspects of the invention are integrated, the part design model may have been previously created (e.g., using CAD processing module **932** of FIG. **12**) and stored (e.g., at PDM data **335** of FIG. **8**). In that case, step **4080** of uploading the part design model may not be required because it is already stored on the system in a neutral format.

In step **4100**, GMR server system **1000** parses the part design model received from a designer into its constituent graphical and non-graphical entities that may be tagged and stored into an object-oriented database, such as the that depicted as stored designer part design model data module **3600** of FIG. **22**. The designer may then assign attributes (and attribute values) which may be associated with the entities stored with the part design model, as in step **4120**. The step of attribute assignment may be carried out by a software component such as that depicted as module **2550** of FIG. **21**. As with step **4080**, in the integrated embodiment of the invention, the complete, attribute-assigned part design model may already be stored, in which case step **4120** is not necessary.

Once the stored part design model is sufficiently complete, in step **4140**, producibility analysis (details of which are provided above in connection with module **2650** of FIG. **21**) may be performed to generate a second query set. As indicated above, the producibility analysis results may be considered intermediate to the extent they are primarily used for generating the second query set; however, the raw results and underlying methodology may be made available for the designer's consideration.

Step **4160** provides for the designer to select preference input criteria in order to conduct a search based on general parameters, as previously discussed. Step **4160** may be performed by a software component such as interactive designer preference module **2400** of FIG. **21**, and may include the provision of a Java™ applet to facilitate preference selection. Based on the preference input criteria, a first query set for searching the database may be generated by GMR server system **1000**, as in step **4180**.

Once these query sets (e.g., the first query set in the case of a limited search and both the first and second query sets in the case of a full search) are generated, those sets are compared to the GMR data, as in step **4200**. The query sets may be compared to all of the fabricator data, as depicted as GMR data module **3000** in FIG. **22**, or only a subset of such data. In the case of a full search, two sets of results may be obtained, in which case step **4200** may include the application of reconciliation logic to combine the results into a single set (e.g., see FIG. **21**, module **2700**). Of course, as should be appreciated by those of ordinary skill in the art, the need for reconciliation logic may be obviated by simply reporting both sets of results.

In step **4220**, the search results for the limited/full search may be saved into designer account data module **3800** (e.g., reporting qualified fabricators) and fabricator account data module **3850** (e.g., providing a referral describing the designer for which each fabricator has been qualified). Similar to step **4380**, step **4240** provides for the query results to be sent to the designer, which may simply encompass an e-mail informing the designer that the results are posted and available for retrieval. Other notification may also be provided. Similarly, step **4260** provides for the referral emanating from the search to be forwarded to qualified fabricators, which may also encompass an e-mail notice.

Steps **4280** and **4400–4440** are repeated substantially as previously described.

A Method for a Designer to Retrieve Reports Using the GMR System

According to an embodiment of the present invention, a method for a designer to retrieve reports from the interactive GMR system is described in FIG. **24**. Referring to FIG. **24**, the designer may log on according to step **5000**, and may be authenticated according to step **5020**. Next, the designer selects the type of report to be generated, as in step **5040**. According to this embodiment, the designer may request a "standard" report detailing the query results from selected search sessions ("Query Results"), or a "custom" report summarizing transactions in which the designer has engaged ("Designer Custom Report"). In one embodiment, source data for both options may be stored with the designer account data, such as in designer account data module **3800** in FIG. **22**.

If the designer wishes a standard report of query results, the designer selects the desired query results from the designer account, as in step **5060**. In one embodiment, one or more browser pages listing past search sessions may be sent to designer user system **1200**, along with various report format options, so that the designer may select and transmit the desired options back to GMR server system **1000**. According to step **5080**, GMR server system **1000** then retrieves the selected data from the designer account and formats the report as requested. According to step **5100**, the report may be then transmitted to the designer. In one embodiment, the designer may be given the option of viewing the report directly as browser pages or of linking to a file for download as a standard word processing text document (e.g., in Microsoft Word™, Word Perfect™, etc.). The designer account data may be updated to reflect this transaction, as in step **5120**.

If the designer desires a custom report, then the designer selects custom report options according to step **5140**, which permits a search of the designer's entire account, including session records, transaction records, and billing records, for example.

According to one embodiment, step **5140** may include the use of a Java™ applet sent from GMR server system **1000** to facilitate the custom reporting option, which may include report format options. Based on the custom report options selected by the designer, GMR server system **1000** generates an appropriate query set, according to step **5160**, which may then be run against the designer account data, according to step **5180**. The results of the query may be retrieved and formatted for the report, according to step **5200**. The report may then be transmitted to the designer, pursuant to step **5220**, which, in one embodiment, entails browser pages and/or a link to a downloadable file. As before, the designer account data may be updated, as in step **5240**.

A Method for a Fabricator Session Using the GMR System

According to an embodiment of the present invention, a method for a fabricator session using the interactive EDM system is described in FIG. **25**. Referring to FIG. **25**, the fabricator logs on to initiate the session, as in step **6000**, and GMR server system **1000** authenticates the fabricator, as in step **6020**. The fabricator selects an option, as in step **6040**, which may include a "Generate Report" option, "Update Database" option, or "General Search" option.

If the fabricator selects the "General Search" option, the steps performed may be similar to those for a designer's general search (see FIG. **23**, steps **4300–4380**). The fabricator selects input criteria for the search of non-proprietary data, according to step **6160**; GMR server system **1000** generates an appropriate query set, according to step **6180**; the query set may then be run against general access fabricator data, according to step **6200**; the search results may be stored in the fabricator account, according to step **6220**; and the search results may be sent to the fabricator, according to step **6240**; and the fabricator account data may be updated to reflect the search session, according to step **6260**.

The "Update Database" option may be selected if the fabricator wishes to update he fabricator data stored in GMR

data module **3000**. Accordingly, step **6060** provides for the fabricator to select the database components that are to be updated. For the GMR database allocation depicted as GMR data module **3000** in FIG. **3**, the fabricator might select, for example, the process machinery group data stored in module **3200** to reflect the purchase of a new machine. According to step **6080**, the fabricator may then update the selected database component with the appropriate information, which may be then sent to back to GMR server system **1000**. In one embodiment, GMR server system **1000** may facilitate the update by sending browser pages to the fabricator containing the existing information for that component, which the fabricator may then amend before selecting a send command on the browser page. Step **6100** provides for GMR server system **1000** to save the updated fabricator information to the appropriate location in the database (e.g., see FIG. **22**, process machinery group data module **3200**). Once the database has been successfully updated, a confirmation notice may be sent to the fabricator, as in step **6120**. Finally, the fabricator account data may be updated to reflect the update session, as in step **6140**.

The "Generate Report" option may be selected if the fabricator wishes to view a report. In this embodiment, three categories of fabricator reports may be supported, as reflected by the three paths emanating from decision block of step **6160**. If the fabricator wishes to review stored fabricator information in the database to ensure it is current, then the "Fabricator Information Report" option may be selected. Accordingly, GMR server system **1000** retrieves the stored fabricator data (e.g., data for that fabricator ID contained in GMR data module **3000**, FIG. **22**) and formats the report, as in step **6180**. The report may then be transmitted to the fabricator, as in step **6200**, which, in one embodiment, may be accessed either as platform independent browser pages or a link to a formatted text file. Finally, the fabricator account data may be updated to reflect the transaction, as depicted by step **6220**.

It is noted that the "Generate Report" and "Update Database" options for a fabricator session represent one embodiment of supporting the so-called "capability management system" previously mentioned. That is, the stored fabricator data on the GMR database is not only used to facilitate searches by designers hoping to identify qualified fabricators, but it incidentally also provides a ready means for fabricators to manage their resources and information. This provides a substantial benefit to fabricators listed in the GMR registry because it obviates or greatly reduces the need for using internal hardware, software and human resources for capability management. This provides a substantial benefit to the GMR provider and participating designers because it may induce fabricators to "enroll" in the registry in the first instance. Thus, fabricators otherwise not inclined to list with the registry may do so because of the benefit of the substantially cost-free capability management resource they will receive. These are significant benefits. In fact, a preferred embodiment may present options for viewing/modifying fabricator data explicitly as the "Manufacturer Capability Management System," so as to highlight this beneficial aspect of the GMR system. In presenting it as such, the GMR provider may also highlight the fact that it is network-based (e.g., "Internet Accessible" or "World Wide Web Accessible" or the like) and secure (e.g., "All Proprietary Information Securely Maintained").

The "Fabricator Referrals Report" option may be selected if the fabricator wishes to view referral results saved in the fabricator account data. The fabricator selects the desired referral results from the fabricator account, as in step **6240**, which, in one embodiment, may be via browser pages sent from GMR server system **1000** listing all referrals in the account. Based on the fabricator's selection, GMR server system **1000** then retrieves the appropriate referral results and formats the report, as in step **6260**. The report may then be sent to the fabricator, as in step **6280**. The fabricator account data may then be updated to reflect the transaction, as in step **6300**.

The "Fabricator Custom Report" option may be selected if the fabricator desires a customized report from the fabricator's account data. This option permits a search of the fabricator's account, similar to that for a designer custom report (discussed above). The fabricator selects the report options, according to step **6320**, which, in one embodiment, may be facilitated through a server-supplied Java™ applet. Thereafter, GMR server system **1000** prepares a query set based on the selected options, as in step **6340**, which may be run against that fabricator's account, as in step **6360**. GMR server system **1000** retrieves the data and formats the report, as in step **6380**, transmits the report to the requesting fabricator, as in step **6400**, and updates the fabricator account data to reflect the transaction, as in step **6420**.

A Third Aspect of the Invention: A System and Methods for Providing an Electronic Trading Community for a Part to be Built

The third aspect of the invention provides an electronic trading community (ETC) whereby a prime contractor/designer with a part design model can electronically solicit bids or proposals by issuing a request for quote (RFQ) or request for proposal (RFP) to fabricators. In the preferred embodiment, the ETC capability of the system is integrated with the first and second aspects of the invention, the NICECAD capability for virtual collaborative engineering, and the GMR capability for identifying qualified fabricators. In that embodiment, a prime contractor may utilize the NICECAD capability to create a part design model, stored at a server in a neutral format, in the virtual engineering environment. The prime contractor may then utilize the GMR capability to identify a pool of qualified fabricators. The prime contractor may then utilize the third aspect of the invention to solicit bids from this pool, or from a subset of the pool if the prime contractor decides to exclude certain fabricators for some reason.

FIG. **26** provides a high-level functional diagram of the ETC capability of the system, including GMR user **2600**, GMR provider **2650** and manufacturing vendors **2680**. GMR user **2600** represents a designer or prime contractor seeking to out-source the manufacture of a product for which there exists a part design model. GMR user **2600** will select vendors for participation in the RFQ/bid process, as indicated by block **2602**. As noted, in the integrated system, GMR user **2600** may select all or a part of the list of qualified fabricators emanating from a GMR search according to the second aspect of the invention. In an alternative embodiment, the ETC RFQ/bid process may be undertaken with one or more manufacturing vendors not previously registered or otherwise associated with the system. In such a case, GMR user **2600** may provide identification and other information to GMR provider **2650** so that the fabricator(s) can access the system to participate.

GMR user **2600** submits a request for quote to GMR provider **2650**, the request comprising design and specification data, such as part design model data, and associated manufacturing specifications, as depicted in block **2604**. If the part design model was created and stored using the NICECAD aspect of the system, the part design model and specification data can be provided by giving access permis-

sions to specified fabricators, as discussed previously in connection with stored design and analysis access permission data module **860** of FIG. **8**. Based on the bids submitted, GMR user **2600** will select one or more winning bidders, as in block **2606**. Upon award, GMR user **2600** submits a complete part definition to the winning bidder(s), as in block **2608**. We call what is eventually provided a "complete" part definition because what is initially provided may be a lower-fidelity data set (e.g., lower resolution graphical data) which is adequate for purposes of the bidding process. Upon completion of the bidding process, complete or high fidelity data is provided to the selected bidder (the "winner") who will actually be fabricating the part.

GMR provider **2650** represents the server-based system providing the ETC capability and communicating with GMR user **2600** and manufacturing vendors **2680** through a secure networked interface. GMR provider **2650** supports so-called transaction mediation (block **2652**) between a prime contractor and bidding manufacturer(s) by supporting terms and conditions negotiations **2654**, contract formalization **2656** and digital signatures **2662**, electronic funds transfer **2658**, and multimedia services **2669**. Transaction mediation generally occurs after a prime contractor selects one or more winning bidders, and involves the formalization of an agreement corresponding to the selected bid. In this context, multimedia services **2669** refers to communications functionality supporting quasi-real time audio, video, and graphics, including CAD functionality supporting viewing and manipulation of a part design model as previously discussed.

Manufacturing vendors **2680** represents fabricators bidding in response to an RFQ or RFP submitted by GMR user **2600**. Manufacturing vendors **2680** may view 3D part design models and associated specifications, as in block **2682**. The ETC system may permit a prospective bidder to view the pool of bidders before preparing and submitting (or revising/updating) a bid, as illustrated by block **2684**. For example, in one embodiment a "bidding pool list" identifying the dollar amount of the various bids is available to the bidders. The bidding pool list does not identify the bidders or disclose any detailed information provided by them to GMR user **2600**. This list might indicate: "Bidder A: $25,000.00; Bidder B: $32,000.00, . . . " and so forth. Ultimately, a successful bidder will product and deliver parts, as in block **2686**.

FIG. **27** further illustrates how the capabilities of the present system are used by a bidding fabricator in evaluating an RFQ. GMR graphics server **2710** (e.g., GMR server system **1000**, FIG. **20**) provides for manufacturing vendor engineer and craftsmen (**2770**) to view the stored part design model and associated features via, in the preferred embodiment, a Java™-enhanced browser (block **2760**). Using the aforementioned CAD capabilities, vendor engineers and craftsman **2770** can examine the part design model according to various views. As depicted by block **2750**, the preferred embodiment utilizes a Java™ 3D version of the part design model supporting manipulation and rotation, pans, zooms and fly-throughs. Manipulation may comprise a request to view the part design model in a higher fidelity representation. As depicted by blocks **2740** and **2720**, graphical entities of the part design model may have links to specification data, standards and materials data sheets. Further regarding block **2720**, and as previously discussed in connection with multimedia communications processing module **978** (FIG. **13**) and CAD processing module **932** (FIG. **12**), so-called "markups" can be attached to the part design model to facilitate discussions with the prime contractor. As discussed previously in connection with the graphics and CAD capabilities of the invention, the viewing and the various manipulations of a part design model (and associated features) could be accomplished by means other than Java™-enhanced browsers. Custom browsers could be used. Also, server-provided semi- machine-dependent applications could be used as well.

FIG. **28** illustrates that the ETC aspect of the invention supports interactive graphics manipulation, similar to that previously discussed in connection with the multimedia communications processing module **978** (FIG. **13**) and CAD processing module **932** (FIG. **12**). GMR graphics server **2810** permits prime contractor design engineers **2812** and manufacturing vendor engineers and craftsmen **2814** to interface via enhanced browsers (blocks **2808** and **2806**) to engage in a quasi- real-time interactive session supporting pan, zoom and fly-throughs, posting of new versions, manipulation and rotation (collectively, block **2802**), and the attachment and viewing of specification data, markups and links to applicable standards (collectively, block **2804**). Again, it bears reiteration that such interface means could be other than Java™-enhanced browsers, including custom browsers and/or server-provided applications executed using standard operating systems. The network interface between users and a server(s) is maintained as substantially platform independent. It is also noted that other engineering data logically associated with features of the part design model may be viewed as well. The various types of specification, standards, and other engineering data that may be logically associated with a part design model was discussed earlier in detail.

Collectively, FIGS. **26–28** illustrate how the networked CAD and multimedia communications features of the first aspect of the invention (NICECAD) can be beneficially applied to support the ETC RFQ and bidding process. In sum, the quasi-real-time "streaming" of audio, video and graphics through a GMR provider provides a means for the prime contractor and bidder to engage in detailed technical discussions from different locations, in a secure environment, and substantially without the need for specialized hardware and software. It can be appreciated that numerous aspects of the so-called "boundaries" difficulty, previously described, are ameliorated or eliminated.

Continuing with the ETC aspect of the invention, the RFQ may be submitted or "posted" to GMR graphics server **2710** via browser templates completed by GMR user **2600**. The RFQ may include such information as a project identifier, narrative description, quantity requirements, schedule requirements, delivery requirements, and the like. The RFQ may include information pertaining to how many "rounds" of bids will be considered, or as to whether a given round is a so-called "best and final" offer round. The RFQ may include information identifying the bidding pool. When the second (GMR) and third (ETC) aspects of the invention are integrated, the RFQ form may include a listing of the qualified fabricators identified by a search. Prime contractor GMR user **2600** may then select from the list so that the RFQ is directed to the selected fabricators. The RFQ may include information identifying the date and time for the bidding round(s), such as indicating the bidding window begins at 8:00 a.m. on November 7 and ending at 4:30 p.m. on November 14. The RFQ may also include information identifying that period when the part design model (and associated) data will be available and/or when GMR user **2600** will be available for conducting multimedia communications sessions to discuss issues.

As previously mentioned, the bidder may, and usually will, view and examine a graphical representation of the part

design model (and associated technical documents or data) before submitting a bid. The bidder may also engage in a multimedia communications session with the soliciting entity to resolve any technical or business issues. A bidder may also view the bidding pool list, previously described, in order to evaluate the current state of the bidding process for the RFQ before submitting or updating a previously-submitted bid.

Bids that are submitted to the GMR provider (e.g., GMR server system **1000**, FIG. **20**) are posted, along with any supplemental information provided by bidders. This supplemental information could provide further information about the bidder or qualifying or clarifying the bid. For example, the bidder might say "Our bid is slightly higher than others because XYZ Machine and Tool, Inc. pays its operators a higher wage so as to retain the highest quality personnel" or other such information to be evaluated by the issuer of the RFQ. This bid and supplemental information may be stored by the GMR provider at a location in database **1100** (FIG. **22**), such as in fabricator account data module **3850**. Generally, GMR server system **1000** may tag the bids with time-stamp data and data identifying the prime contractor to which the bid is directed. After evaluating the bids, a GMR user selects the vendor(s) best meeting the RFP requirements, and winning bidders are notified.

Once the bidding window is closed, the issuer of the RFP/RFQ evaluates the bids and supplemental information. Vendors selected for contract negotiation and contract award are notified as a process of contract formalization commences. A collection of standard clauses are stored in a database at one of the NICECAD/GMR provider's servers. These clauses outline the responsibilities of each party with respect to common aspects of parts procurement. An example might be the specific assignment of responsibility for part inspection and verification. A clause may be provided that assigns this responsibility to the fabricator or to the prime contractor depending on the outcome of contract negotiations. Since many similar situations exist in this type of buying, standard clauses may be used as bases for the formal contract. Such standard contract clauses serve as starting points for negotiation. Through the virtual environment, these contract clauses are modified and included in a "formalized" contract that is custom tailored for the specific manufacturing job. A formalized contract may be thought of as a "signable" contract that has resulted from negotiations over a set of "standard" starting points. This feature is necessary to accommodate the very dynamic nature of this type of procurement.

The terms and conditions negotiations may use the secure multimedia communications capability previously discussed, whereby a prime contractor and successful bidder can engage in negotiations using the quasi- real-time video and/or audio. Stored standard form contracts (see, e.g., module **696**, FIG. **6**) and stored standard terms and conditions (see, e.g., module **697**, FIG. **6**) may be employed. The digital signature capability provides a means for both parties to formalize agreements once they are negotiated (see, e.g., module **699**, FIG. **6**). Once negotiated and formalized, contracts may be stored by the GMR provider, such as at module **415** of FIG. **4**.

A final aspect to the present invention is that the provider the GMR provider (e.g, GMR server system **1000**) may further provide to the winning fabricator(s) a network-based connection to the prime contractor's Enterprise Resource Planning (ERP) system. ERP generally refers to a multi-module software application integrated with a relational database used by a prime contractor to manage important parts of its business, such as product planning, parts purchasing, inventory management, interaction with suppliers, customer service and order tracking. By providing a fabricator, (or other supplier) a secure connection (or so-called "portal") to the prime contractor's ERP system, the fabricator is further integrated with the prime contractor and business boundaries are further diminished. Commercially available ERP applications are available from PeopleSoft, SAP, Oracle, and Baan.

Other embodiments and uses of this invention will be apparent to those having ordinary skill in the art upon consideration of the specification and practice of the invention disclosed herein. The specification and examples given should be considered exemplary only, and it is contemplated that the appended claims will cover any other such embodiments or modifications as fall within the true scope of the invention.

What is claimed is:

1. A server-based system for a fabricator evaluating detailed manufacturing instructions contained within a request for a propsal to view a part design model, comprising:

a memory for storing a part design model provided by a designer seeking a proposal for manufacturing the part represented by the part design model, said memory resident at a first location. said designer resident at a second location and said fabricator resident at a third location. where the first location is separate from both the second and third locations; and

a server system for enabling a fabricator connected over a packet-switched network to access said part design model;

said server system having a software component for presenting the part design model to the fabricator through said network using a graphical user interface and where said software component is resident at said server system and accessible by said software component includes a substantially planform indenpendent client side application to be run on the system of the fabricator. where said application permits the manipulation of the part design model.

2. The server-based system of claim **1**, wherein said manipulation comprises one or more of rotation, translation, two-dimensional cutting and a fly-through.

3. The server-based system of claim **1**, wherein the part design model comprises a plurality of three-dimensional graphical features which are linked by the server system with at least one of specifications, regulations and standards associated with each of the plurality of features.

4. The server-based system of claim **3**, wherein said software component further provides for the fabricator to select a specific three-dimensional graphical feature in order to view the at least one associated specification, regulations and standard associated with the selected feature.

5. The server based system of claim **4**, wherein the selection is performed using a mouse device.

6. The server-based system of claim **1**, wherein said server system is further adapted to permit the designer to review the accuracy of detailed manufacturing instructions contained within said request for a proposal.

7. The server based system of claim **1**, wherein said software component further enables the fabricator and the designer to engage in a communications session that is substantially real-time.

8. The server based system of claim **7**, wherein said communications session comprises one or more of audio and video.

9. The server based system of claim 7, wherein said communications session comprises the simultaneous presentation of the part design model to the fabricator and the designer.

10. The server based system of claim 9, wherein the simultaneous presentation includes the manipulation of the part design model.

11. The server-based system of claim 4, wherein the at least one associated specifications, regulations and standrds reside in the server memory and are sent over the network connection to the client-side on demand.

12. The server-based system of claim 6, wherein the part design model comprises a plurality of three-dimensional graphical features which are linked by the server system with at least one specification, regulation and standard associated with each feature.

13. The server based system of claim 1, wherein said software component further provides for the fabricator to select a specific three-dimensional graphical feature in order to view the at least one associated specification, regulation and standard.

14. The server-based system of claim 13, wherein the at least one associated specification, regulation and standard reside in the server memory and are sent over the network connection to the client-side application on demand.

15. A server-based system for a fabricator evaluating detailed manufacturing instructions to view a part design model, comprising:

a memory for storing part design model provided by a designer where a part is represented by the part design model, wherein the part design model comprises at least one three-dimensional graphical depiction having a plurality of features which are linked by the server system with at least one specification, regulation, and standard associated with each of the plurality of features; and a server system for enabling a fabricator connected over a packet-switched network to access said part design model:

said server system having a software component for presenting the part design model to the fabricator through said network using a graphical user interface. wherein said software component further provides for the fabricator to select a specific feature of the three-dimensional graphical depiction in order to view the specification, regulation, and standard associated with the selected feature.

16. The server-based system of claim 15, wherein said software component further permits the manipulation of the part design model.

17. The server-based system of claim 15, wherein said manipulation comprises one or more of rotation, translation, two-dimensional cutting and a fly-through.

18. The server-based system of claim 15, wherein the selection is performed using a mouse device.

19. The server-based system of claim 15, wherein said server system is further adapted to permit the designer to review the accuracy of detailed manufacturing instructions contained within said request for a proposal.

20. The server-based system of claim 15, wherein said proposal includes at least a portion of the part design model.

21. The server-based system of claim 15, wherein said part design model was stored in said memory at a time before submission of said proposal.

22. The server-based system of claim 15, wherein said software component further enables the fabricator and the designer to engage in a communications session that is substantially real-time.

23. The server-based system of claim 22, wherein said communications session comprises one or more of audio and video.

24. The server-based system of claim 22, communications session comprises the simultaneous presentation of the part design model to the fabricator and the designer.

25. The server-based system of claim 22, wherein the simultaneous presentation includes the manipulation of the manipulation of the part design model.

26. A process for obtaining information for a fabricators registry comprising the steps of:

providing one or more fabricator manager applications to at least one fabricator, where the one or more fabricator manager applications:

1) assist the at least one fabricator in managing the fabricator'business;

2) reside in a central repository; and

3) are accessed through a global open network.

receiving data from the at least one fabricator, where the data is the result of the at least one fabricator using the one or more fabricator manager applications;

aggregating the received data into a fabricator registry; and

providing access to the fabricator registry to at least one designer, where at least one of:

a) the at least one designer accesses at least a portion of the received data in the fabricator registry; and

b) the entiry providing the fabricator manager applications accesses at least a portion of the received data in the course of providing services to one or more customers.

27. The process according to claim 26, where the one or more fabricator applications comprises a tool inventory application, where the fabricator uses the tool inventory application to maintain a catalog of standard and custom tools used by the fabricator.

28. The process according to claim 26, where the one or more fabricator applications comprises a machine availability scheduling application, where the fabricator uses the machine availability scheduling application to maintain a schedule indicating jobs one or more machines are scheduled for and when the one or more machines are available for a new job.

29. The process according to claim 26, where the one or more fabricator applications comprises a machine maintenance scheduling application, where the one or more fabricator uses the machine maintenance scheduling application to maintain a schedule indicating when one or more machines are scheduled for maintenance and when each of the one or more machines last had maintenance.

30. The process according to claim 26, where the one or more fabricator applications comprises an accounting application, where the fabricator uses the accounting application to maintain billing information, transaction records, and billing records.

31. The process according to claim 26, where the one or more fabricator applications comprises a training management application, where the fabricator uses the training management application to maintain at least one record of the training status of one or more employees.

32. The process according to claim 31, where the training status of the one or more employees includes at least one of:

a) certifications received by the one or more employees;

b) the most recent training of the one or more employees;

c) the next scheduled training for the one or more employees;

d) the years of experience of the one or more employees; and

e) the machines the one or more employees are trained on.

33. The process according to claim 26, where the one or more fabricator applications comprises a certification management application, where the fabrication uses the certification management application to maintain at least one record of the certification status of the fabricator, and to track the compliance status of the fabricator for one or more certifications.

34. The process according to claim 26, further comprising the step of receiving a selection of one or more of the at least one fabricator by the at least one designer based on the accessed data.

35. A system for obtaining information for a fabricators registry comprising:

at least one fabricator module;

an application provider module for providing one or more fabricator manager applications to the at least one fabricator module, where the one or more fabricator manager applications assist the at least one fabricator module in managing the fabricator'business, data is input by the at least one fabricator module using the one or more fabricator manager applications, and the data is aggregated into a fabricator registry; and

at least one designer module, where at least one portion of:

a) the at least one designer module accesses a portion of the data in the fabricator registry; and

b) the entiry providing the fabricator manage application accesses at least a portion of the received data in the course of providing services to one or more customers.

36. The system according to claim 35, where the one or more fabricator applications comprises a tool inventory application. where the fabricator module uses the tool inventory application to maintain a catalog of standard and custom tools used by the fabricator module.

37. The system according to claim 35, where the one or more fabricator applications comprises a machine availability scheduling application, where the fabricator module uses the machine availability scheduling application to maintain a schedule indication jobs one or more machines are scheduled for and when the one or more machines are available for a new job.

38. The system according to claim 35, where the one or more fabricator applications comprises a machine maintenance scheduling application, where the fabricator module uses the machine maintenance scheduling application to maintain a schedule indicating when one or more machines are scheduled for maintenance and when each of the one or more machines last had maintenance.

39. The system according to claim 35, where the one or more fabricator applications comprises an accounting application, where the fabricator module uses the accounting application to maintain billing information, transaction records, and billing records.

40. The system according to claim 35, where the one or more fabricator applications comprises a training management application, where the fabricator module uses the training management application to maintain at least one record of the training status of one or more employees.

41. The system according to claim 35, where the training status of the one or more employees includes at least one of:

a) certifications received by the one or more employees;

b) the most recent training of the one or more employees;

c) the next scheduled training for the one or more employees;

d) the years of experience of the one or more employees; and

e) the machines the one or more employees are trained on.

42. The system according to claim 35, where the one or more fabricator applications comprises a certification management application, where the fabricator module uses the certification management application to maintain at least one record of the certification status of the fabricator module, and to track the compliance status of the fabricator for one or more certifications.

43. The system according to claim 35, where the at least one designer module selects one or more of the fabricator modules based on the accessed data.

* * * * *

US006125391A

United States Patent [19]

Meltzer et al.

[11] Patent Number: 6,125,391

[45] Date of Patent: *Sep. 26, 2000

[54] MARKET MAKERS USING DOCUMENTS FOR COMMERCE IN TRADING PARTNER NETWORKS

[75] Inventors: Bart Alan Meltzer, Aptos; Terry Allen, Sebastopol; Matthew Daniel Fuchs, Los Gatos; Robert John Glushko, San Francisco, all of Calif.; Murray Maloney, Pickering, Canada

[73] Assignee: Commerce One, Inc., Mountain View, Calif.

[*] Notice: This patent is subject to a terminal disclaimer.

[21] Appl. No.: 09/173,854

[22] Filed: Oct. 16, 1998

[51] Int. Cl.[7] G06F 13/00

[52] U.S. Cl. 709/223; 707/513; 705/26; 709/230; 370/466

[58] Field of Search 709/223, 230; 705/26; 707/513; 370/466

[56] References Cited

U.S. PATENT DOCUMENTS

5,742,845	4/1998	Wagner	710/11
6,012,098	1/2000	Bayeh et al.	709/246

FOREIGN PATENT DOCUMENTS

0 704 795 A1	of 1996	European Pat. Off.	G06F 9/44

OTHER PUBLICATIONS

"W3C: Extensible Markup Language (XML) 1.0—W3C Recommendation Feb. 10, 1998", http://www.w3.org/TR/1998/REC-xml-19980210, Printed from Internet Feb. 17, 1998, pp. 1–37.

Kimbrough, et al., "On Automated Message Processing in Electronic Commerce and Work Support Systems: Speech Act Theory and Expressive Felicity", ACM Transactions on Information Systems, vol. 15, No. 4, Oct. 1997, pp. 321–367.

Fuchs, Matthew, "Domain Specific Languages for ad hoc Distributed Applications", USENIX Associate, Conference on Domain-Specific Languages, Oct. 15–17, 1997, pp. 27–35.

Finin, et al., "KQML as an Agent Communication Language", Association of Computing Machinery, 1994, pp. 456–463.

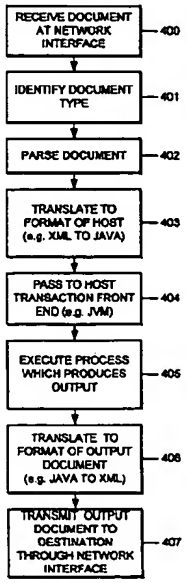
Primary Examiner—Kenneth R. Coulter

Attorney, Agent, or Firm—Wilson, Sonsini, Goodrich & Rosati

[57] ABSTRACT

A market making node in a network routes machine readable documents to connect businesses with customers, suppliers and trading partners. The self defining electronic documents, such as XML based documents, can be easily understood amongst the partners. Definitions of these electronic business documents, called business interface definitions, are posted on the Internet, or otherwise communicated to members of the network. The business interface definitions tell potential trading partners the services the company offers and the documents to use when communicating with such services. Thus, a typical business interface definition allows a customer to place an order by submitting a purchase order or a supplier checks availability by downloading an inventory status report. Also, the registration at a market maker node of a specification of the input and output documents, coupled with interpretation information in a common business library, enables participants in a trading partner network to execute the transaction in a way which closely parallels the way in which paper based businesses operate.

47 Claims, 16 Drawing Sheets



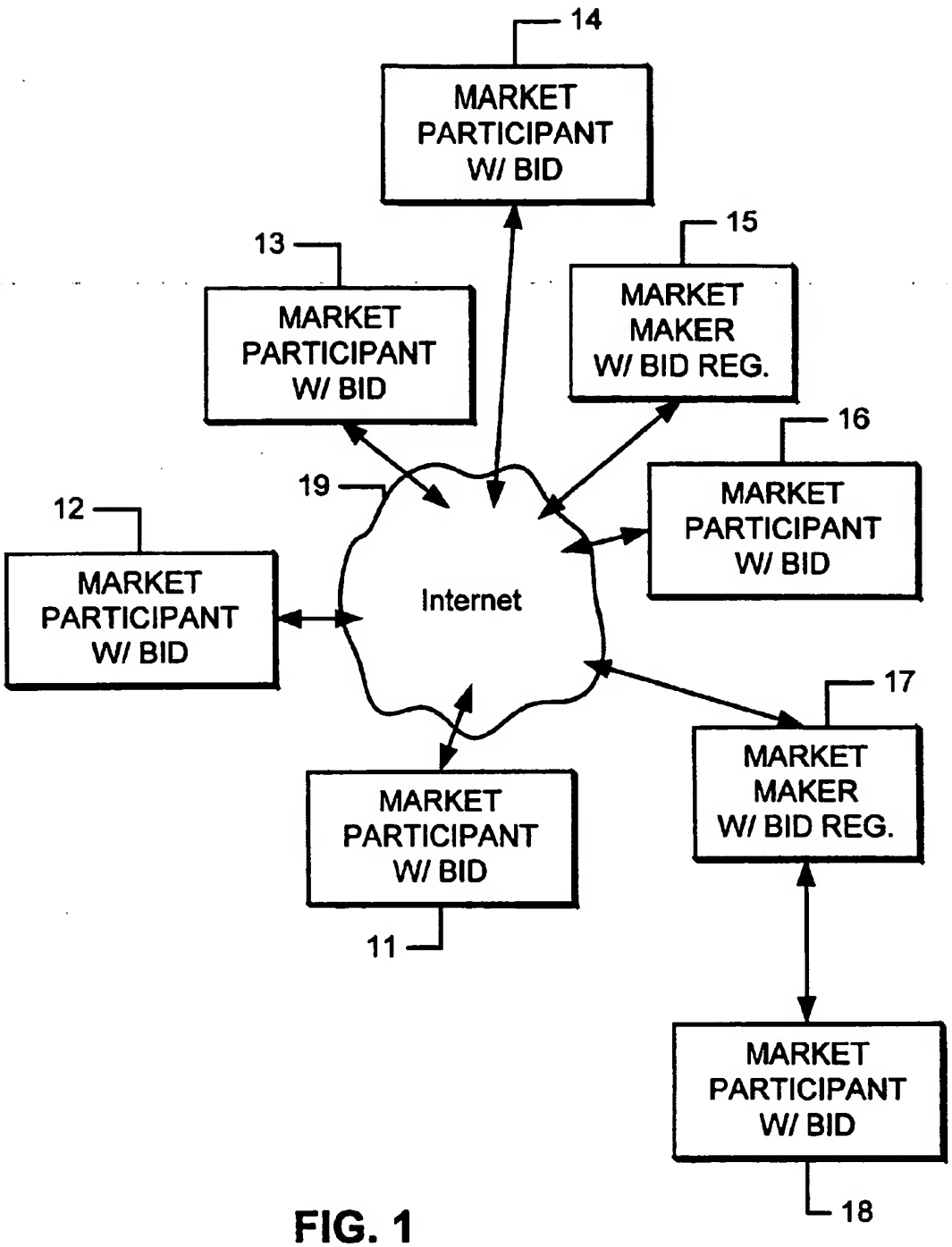
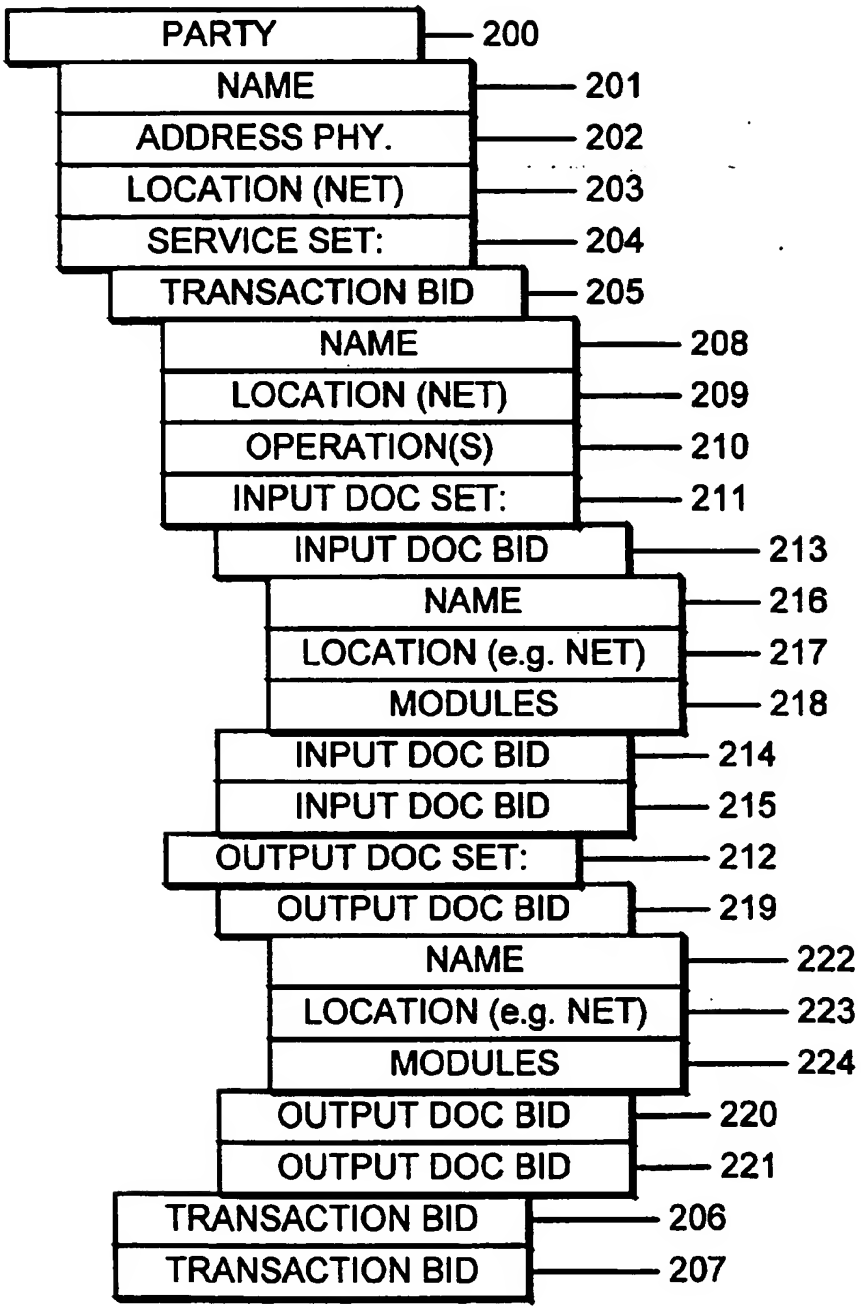

14
MARKET PARTICIPANT W/ BID
13
MARKET PARTICIPANT W/ BID
15
MARKET MAKER W/ BID REG.
16
MARKET PARTICIPANT W/ BID
19
12
MARKET PARTICIPANT W/ BID
Internet
17
MARKET MAKER W/ BID REG.
MARKET PARTICIPANT W/ BID
11
MARKET PARTICIPANT W/ BID
18

FIG. 1

FIG. 2


PARTY
200
NAME
201
ADDRESS PHY.
202
LOCATION (NET)
203
SERVICE SET:
204
TRANSACTION BID
205
NAME
208
LOCATION (NET)
209
OPERATION(S)
210
INPUT DOC SET:
211
INPUT DOC BID
213
NAME
216
LOCATION (e.g. NET)
217
MODULES
218
INPUT DOC BID
214
INPUT DOC BID
215
OUTPUT DOC SET:
212
OUTPUT DOC BID
219
NAME
222
LOCATION (e.g. NET)
223
MODULES
224
OUTPUT DOC BID
220
OUTPUT DOC BID
221
TRANSACTION BID
206
TRANSACTION BID
207

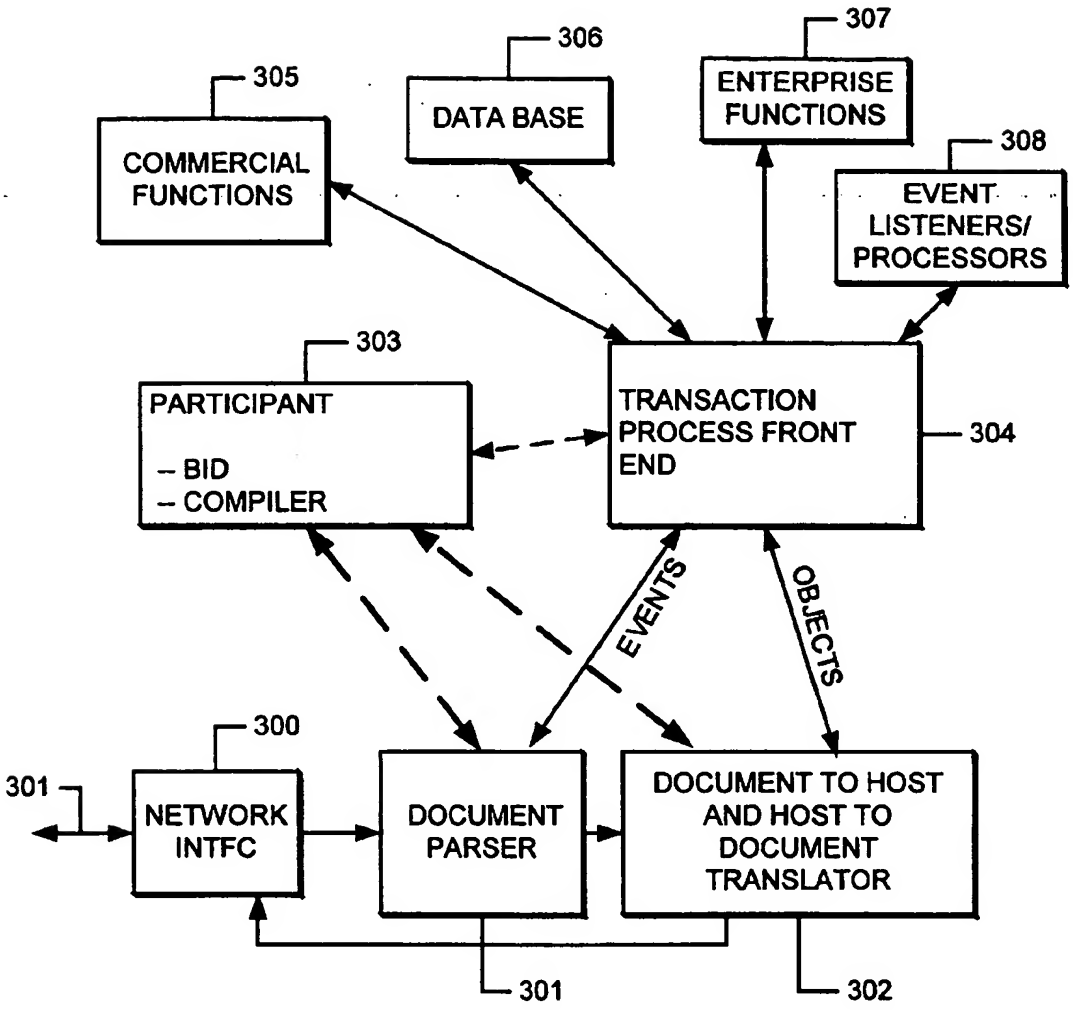

307
306
305
ENTERPRISE FUNCTIONS
DATA BASE
COMMERCIAL FUNCTIONS
308
EVENT LISTENERS/ PROCESSORS
303
PARTICIPANT
– BID
– COMPILER
TRANSACTION PROCESS FRONT END
304
EVENTS
OBJECTS
300
301
NETWORK INTFC
DOCUMENT PARSER
DOCUMENT TO HOST AND HOST TO DOCUMENT TRANSLATOR
301
302

FIG. 3

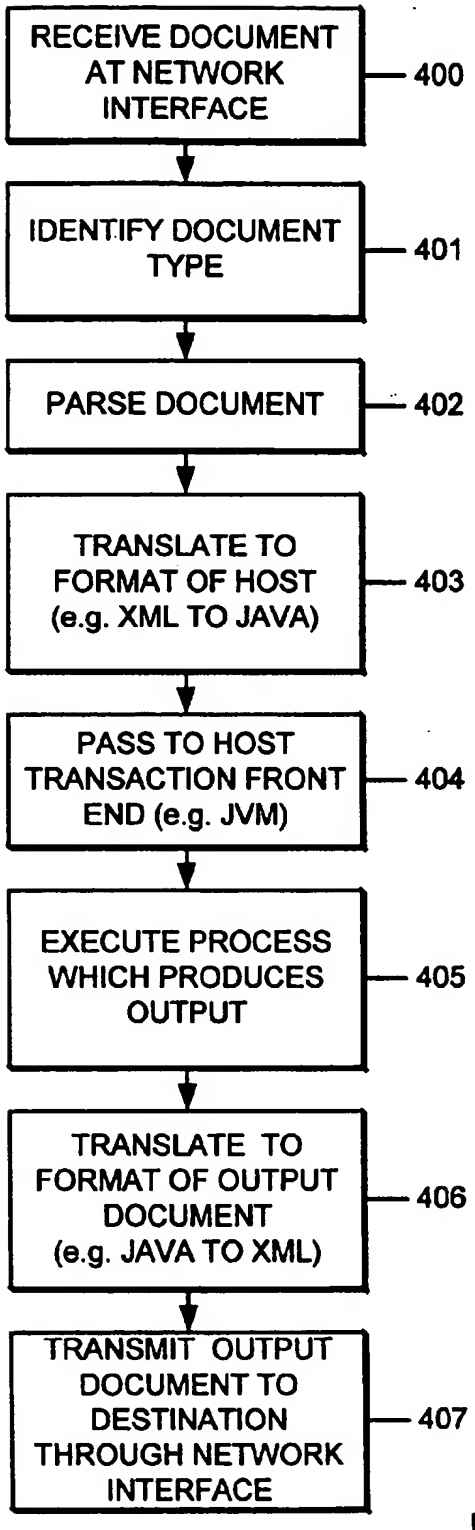

RECEIVE DOCUMENT AT NETWORK INTERFACE
400
IDENTIFY DOCUMENT TYPE
401
PARSE DOCUMENT
402
TRANSLATE TO FORMAT OF HOST (e.g. XML TO JAVA)
403
PASS TO HOST TRANSACTION FRONT END (e.g. JVM)
404
EXECUTE PROCESS WHICH PRODUCES OUTPUT
405
TRANSLATE TO FORMAT OF OUTPUT DOCUMENT (e.g. JAVA TO XML)
406
TRANSMIT OUTPUT DOCUMENT TO DESTINATION THROUGH NETWORK INTERFACE
407

FIG. 4

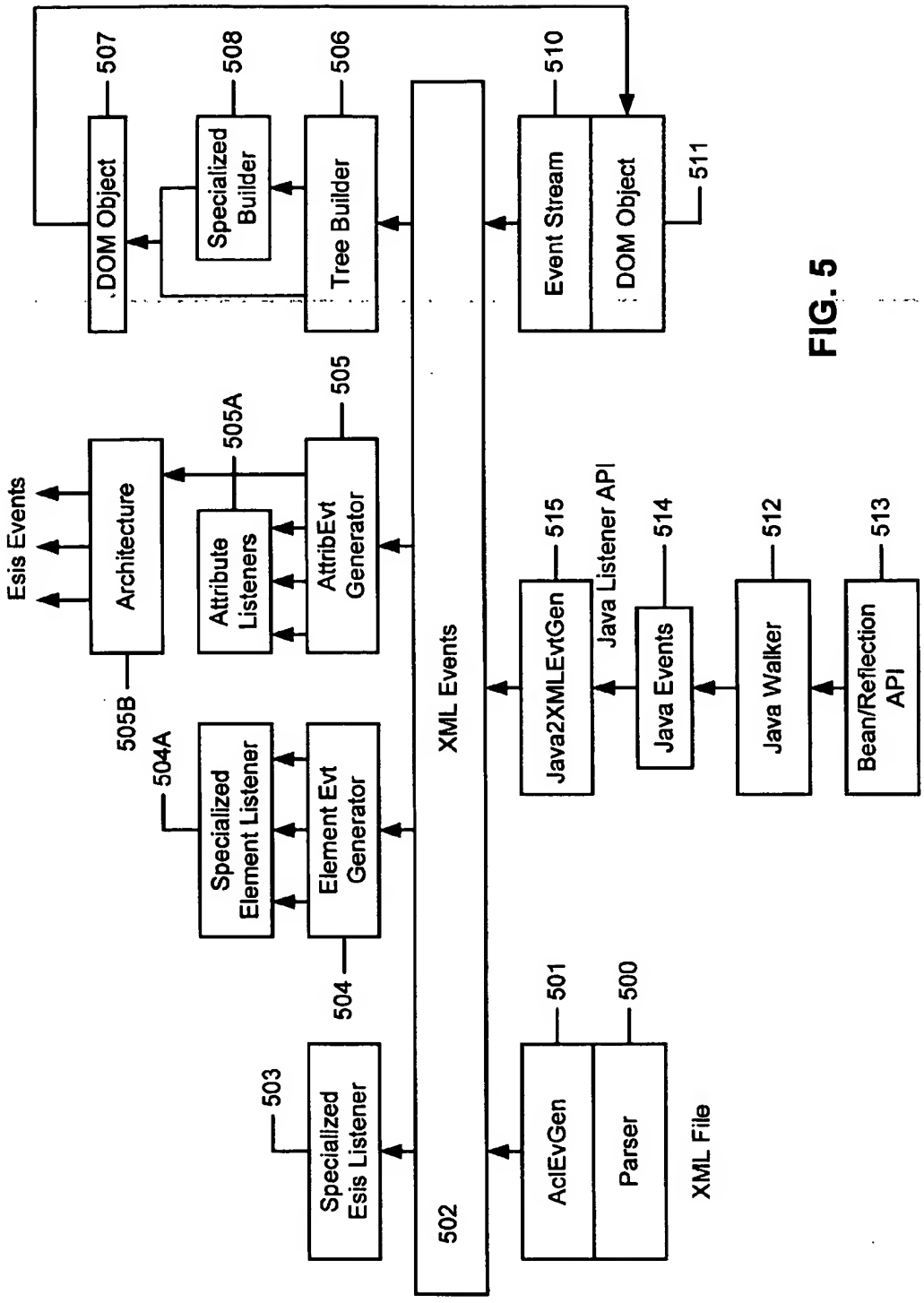

DOM Object
507
Specialized Builder
508
Tree Builder
506
Event Stream
DOM Object
510
511
Esis Events
Architecture
505B
505A
Attribute Listeners
AttribEvt Generator
505
504A
Specialized Element Listener
Element Evt Generator
504
XML Events
Java2XMLEvtGen
515
Java Listener API
Java Events
514
Java Walker
512
Bean/Reflection API
513
503
Specialized Esis Listener
AclEvGen
501
Parser
500
XML File
502

FIG. 5

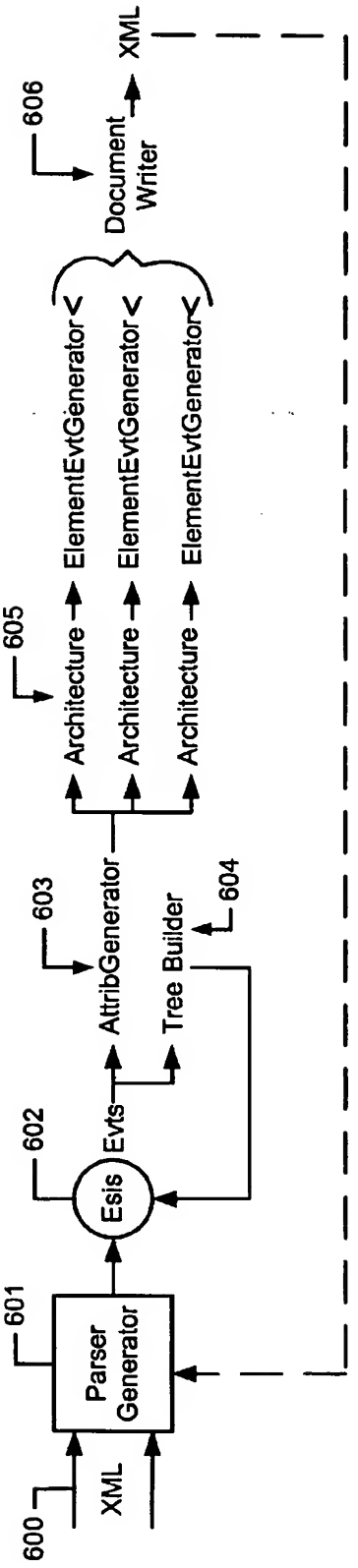

600
XML
601
Parser Generator
602
Esis
Evts
603
AttribGenerator
604
Tree Builder
605
Architecture
ElementEvtGenerator
Architecture
ElementEvtGenerator
Architecture
ElementEvtGenerator
606
Document Writer
XML

FIG. 6

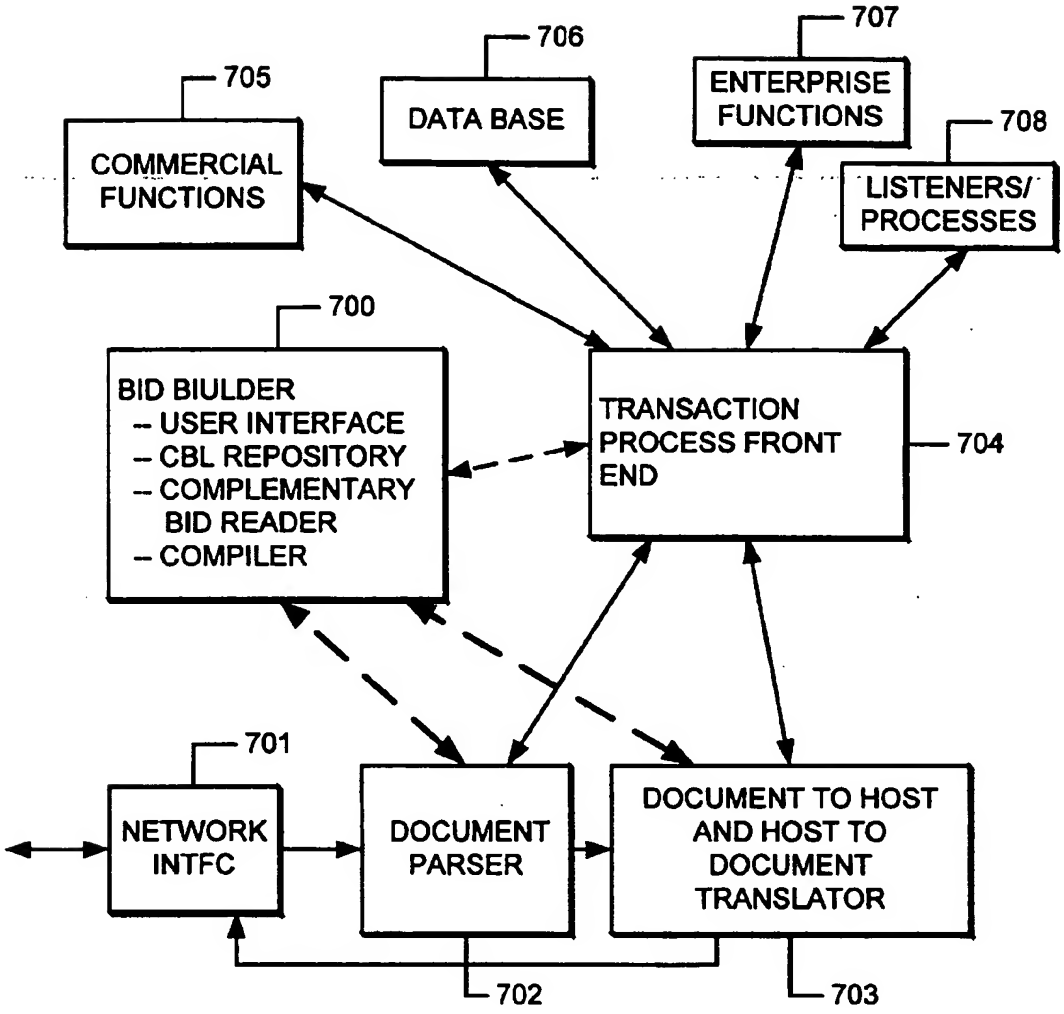

705
COMMERCIAL FUNCTIONS
706
DATA BASE
707
ENTERPRISE FUNCTIONS
708
LISTENERS/ PROCESSES
700
BID BIULDER
-- USER INTERFACE
-- CBL REPOSITORY
-- COMPLEMENTARY BID READER
-- COMPILER
TRANSACTION PROCESS FRONT END
704
701
NETWORK INTFC
DOCUMENT PARSER
DOCUMENT TO HOST AND HOST TO DOCUMENT TRANSLATOR
702
703

FIG. 7

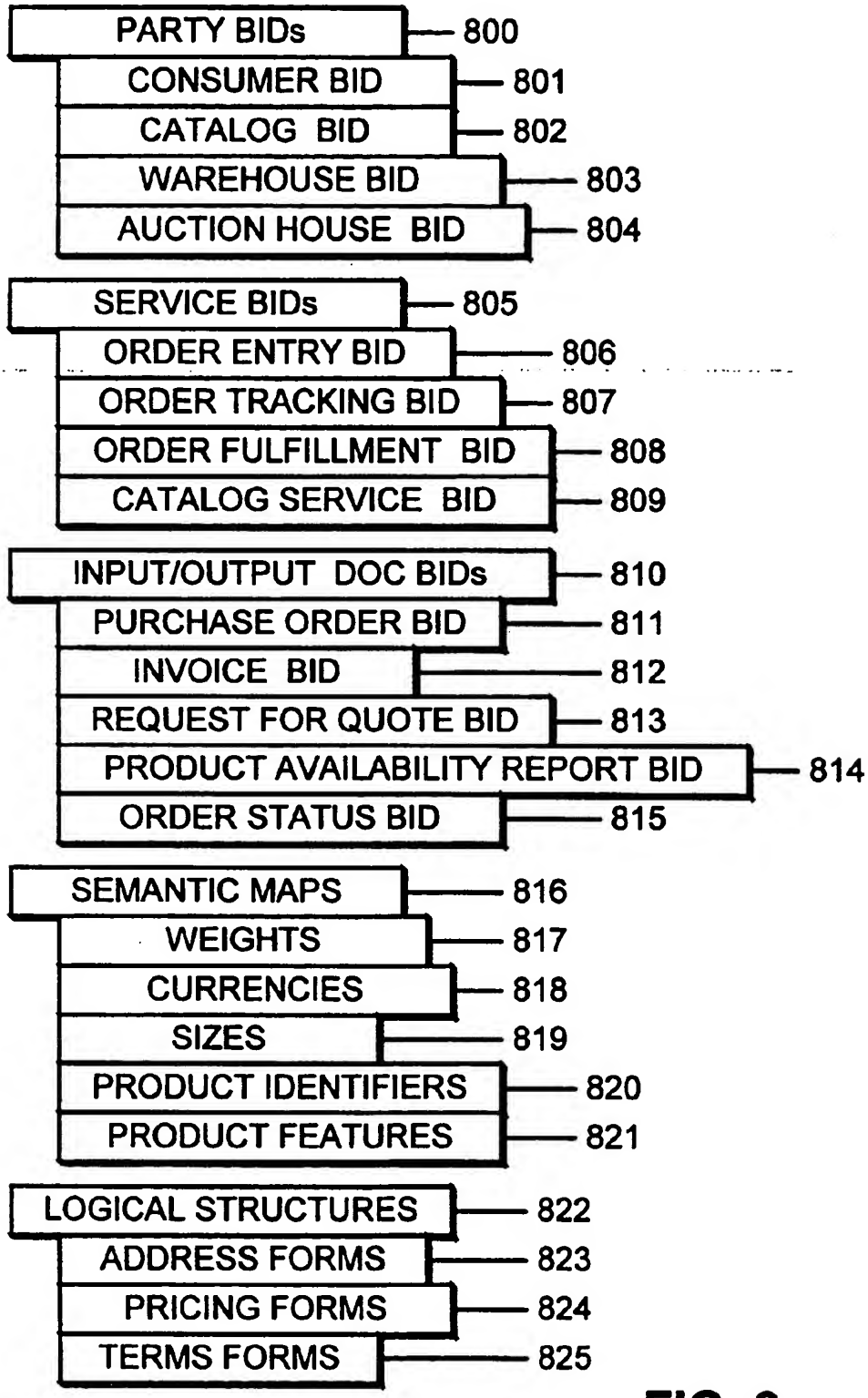

PARTY BIDs
800
CONSUMER BID
801
CATALOG BID
802
WAREHOUSE BID
803
AUCTION HOUSE BID
804
SERVICE BIDs
805
ORDER ENTRY BID
806
ORDER TRACKING BID
807
ORDER FULFILLMENT BID
808
CATALOG SERVICE BID
809
INPUT/OUTPUT DOC BIDs
810
PURCHASE ORDER BID
811
INVOICE BID
812
REQUEST FOR QUOTE BID
813
PRODUCT AVAILABILITY REPORT BID
814
ORDER STATUS BID
815
SEMANTIC MAPS
816
WEIGHTS
817
CURRENCIES
818
SIZES
819
PRODUCT IDENTIFIERS
820
PRODUCT FEATURES
821
LOGICAL STRUCTURES
822
ADDRESS FORMS
823
PRICING FORMS
824
TERMS FORMS
825

FIG. 8

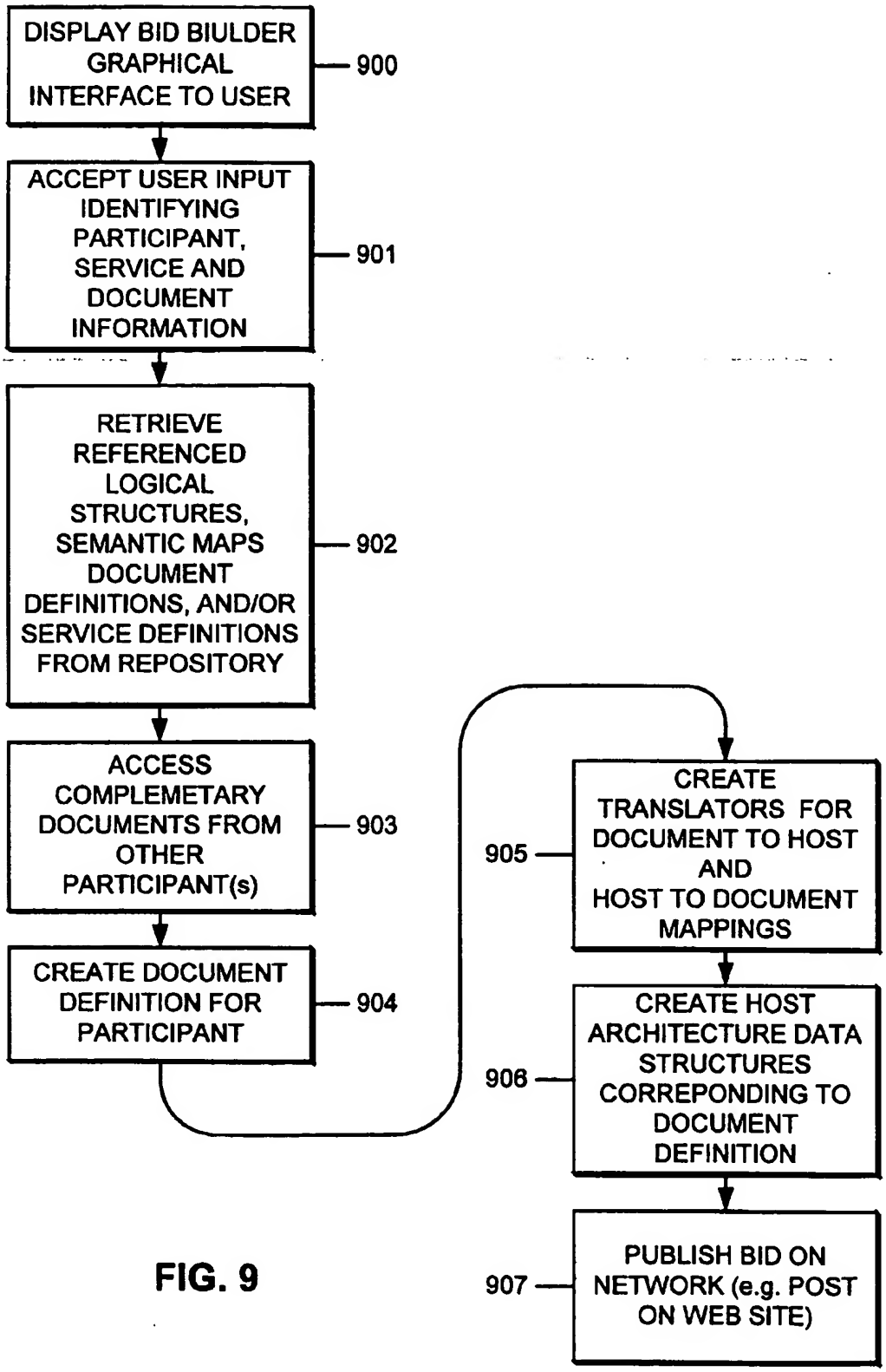

DISPLAY BID BIULDER GRAPHICAL INTERFACE TO USER
900
ACCEPT USER INPUT IDENTIFYING PARTICIPANT, SERVICE AND DOCUMENT INFORMATION
901
RETRIEVE REFERENCED LOGICAL STRUCTURES, SEMANTIC MAPS DOCUMENT DEFINITIONS, AND/OR SERVICE DEFINITIONS FROM REPOSITORY
902
ACCESS COMPLEMETARY DOCUMENTS FROM OTHER PARTICIPANT(s)
903
CREATE DOCUMENT DEFINITION FOR PARTICIPANT
904
905
CREATE TRANSLATORS FOR DOCUMENT TO HOST AND HOST TO DOCUMENT MAPPINGS
906
CREATE HOST ARCHITECTURE DATA STRUCTURES CORREPONDING TO DOCUMENT DEFINITION
907
PUBLISH BID ON NETWORK (e.g. POST ON WEB SITE)

FIG. 9

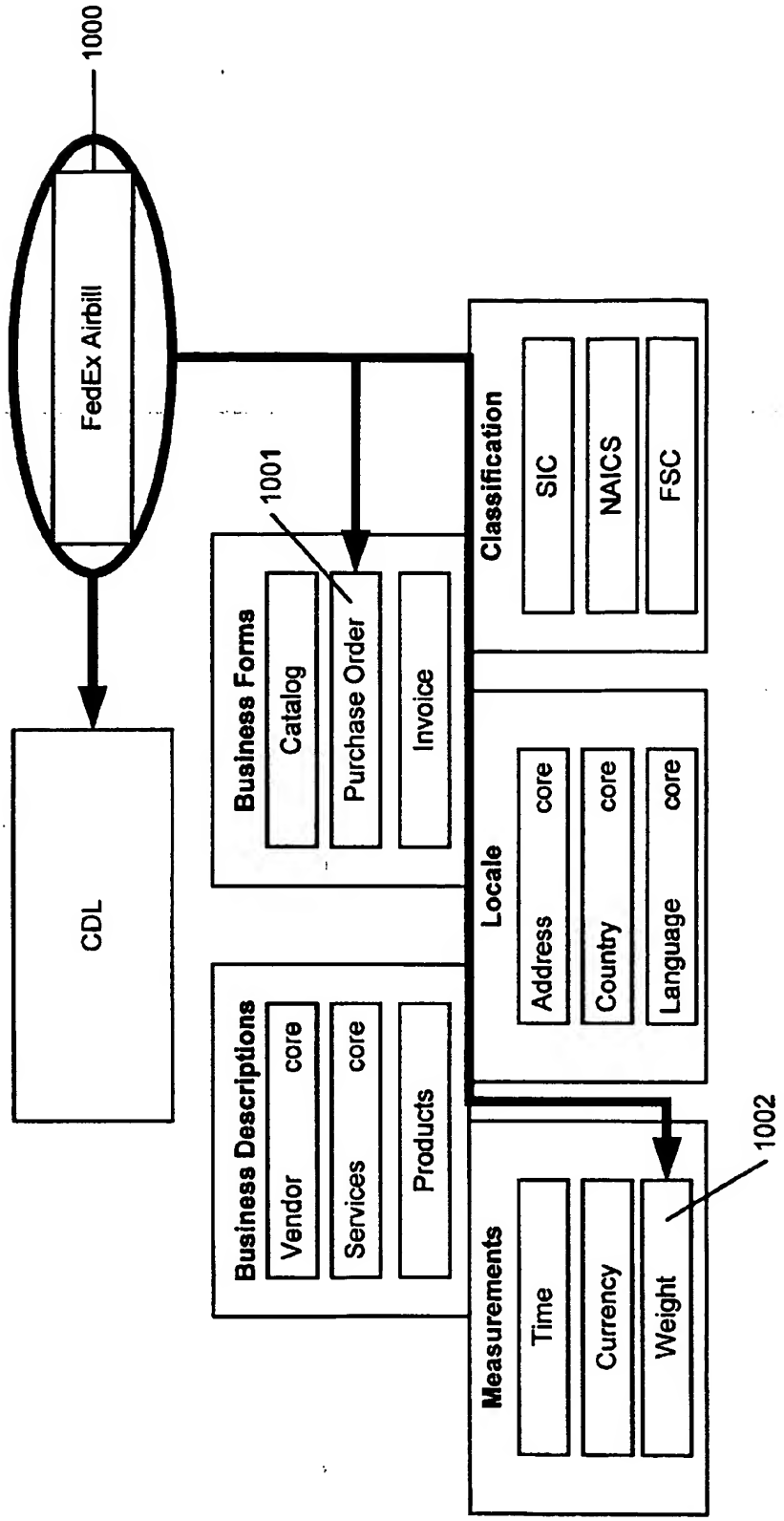

1000
FedEx Airbill
CDL
1001
Business Forms
Catalog
Purchase Order
Invoice
Business Descriptions
Vendor core
Services core
Products
Classification
SIC
NAICS
FSC
Locale
Address core
Country core
Language core
1002
Measurements
Time
Currency
Weight

FIG. 10

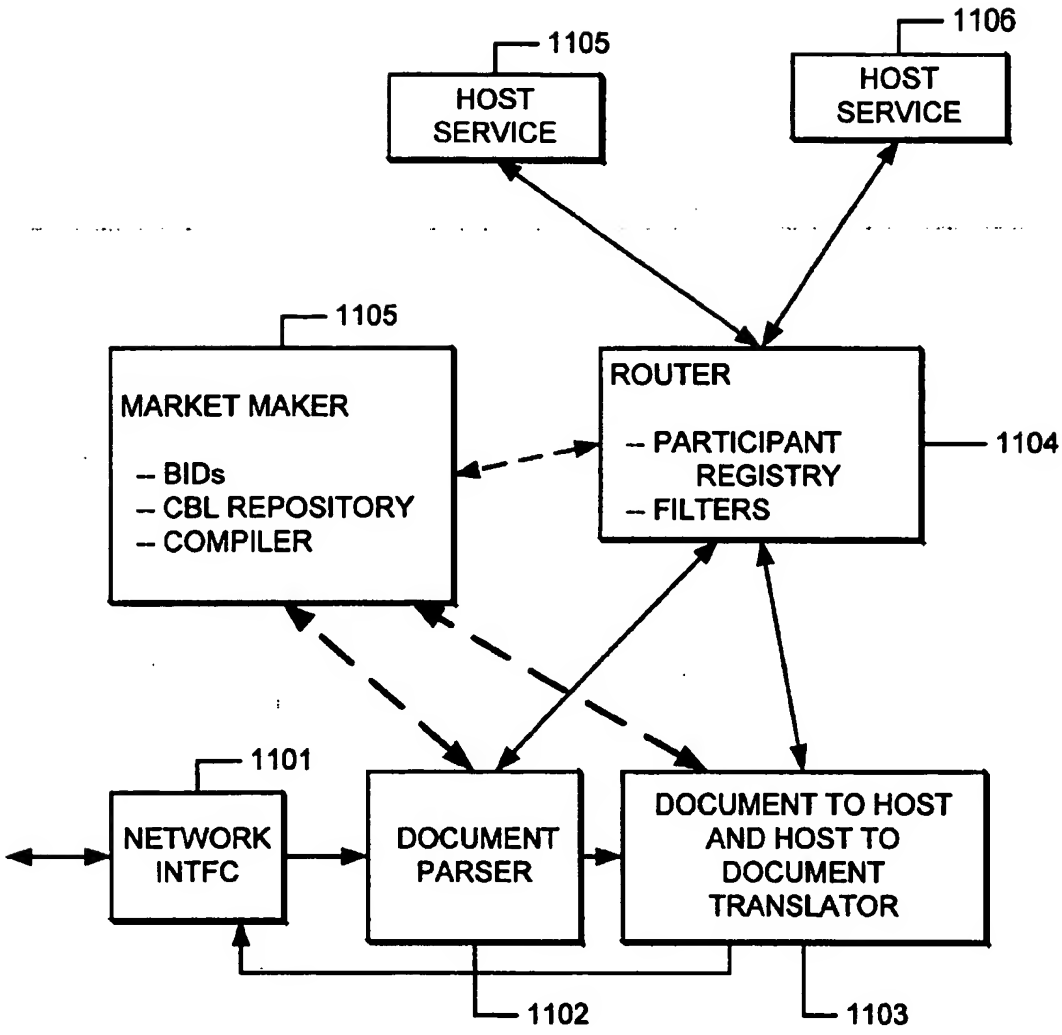

1105
HOST SERVICE
1106
HOST SERVICE
1105
MARKET MAKER
– BIDs
– CBL REPOSITORY
– COMPILER
ROUTER
– PARTICIPANT REGISTRY
– FILTERS
1104
1101
NETWORK INTFC
DOCUMENT PARSER
DOCUMENT TO HOST AND HOST TO DOCUMENT TRANSLATOR
1102
1103

FIG. 11

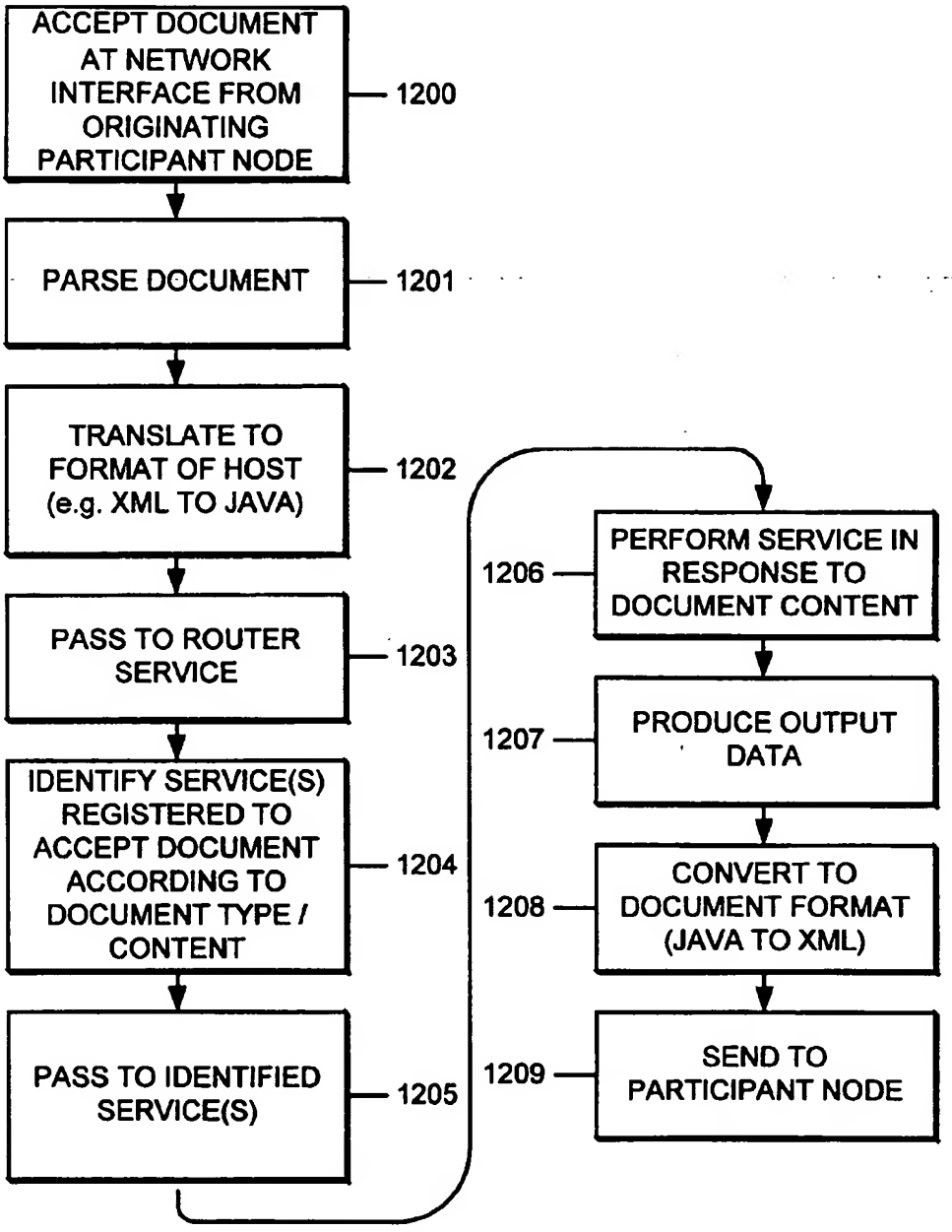

ACCEPT DOCUMENT AT NETWORK INTERFACE FROM ORIGINATING PARTICIPANT NODE
1200
PARSE DOCUMENT
1201
TRANSLATE TO FORMAT OF HOST (e.g. XML TO JAVA)
1202
PASS TO ROUTER SERVICE
1203
IDENTIFY SERVICE(S) REGISTERED TO ACCEPT DOCUMENT ACCORDING TO DOCUMENT TYPE / CONTENT
1204
PASS TO IDENTIFIED SERVICE(S)
1205
PERFORM SERVICE IN RESPONSE TO DOCUMENT CONTENT
1206
PRODUCE OUTPUT DATA
1207
CONVERT TO DOCUMENT FORMAT (JAVA TO XML)
1208
SEND TO PARTICIPANT NODE
1209

FIG. 12

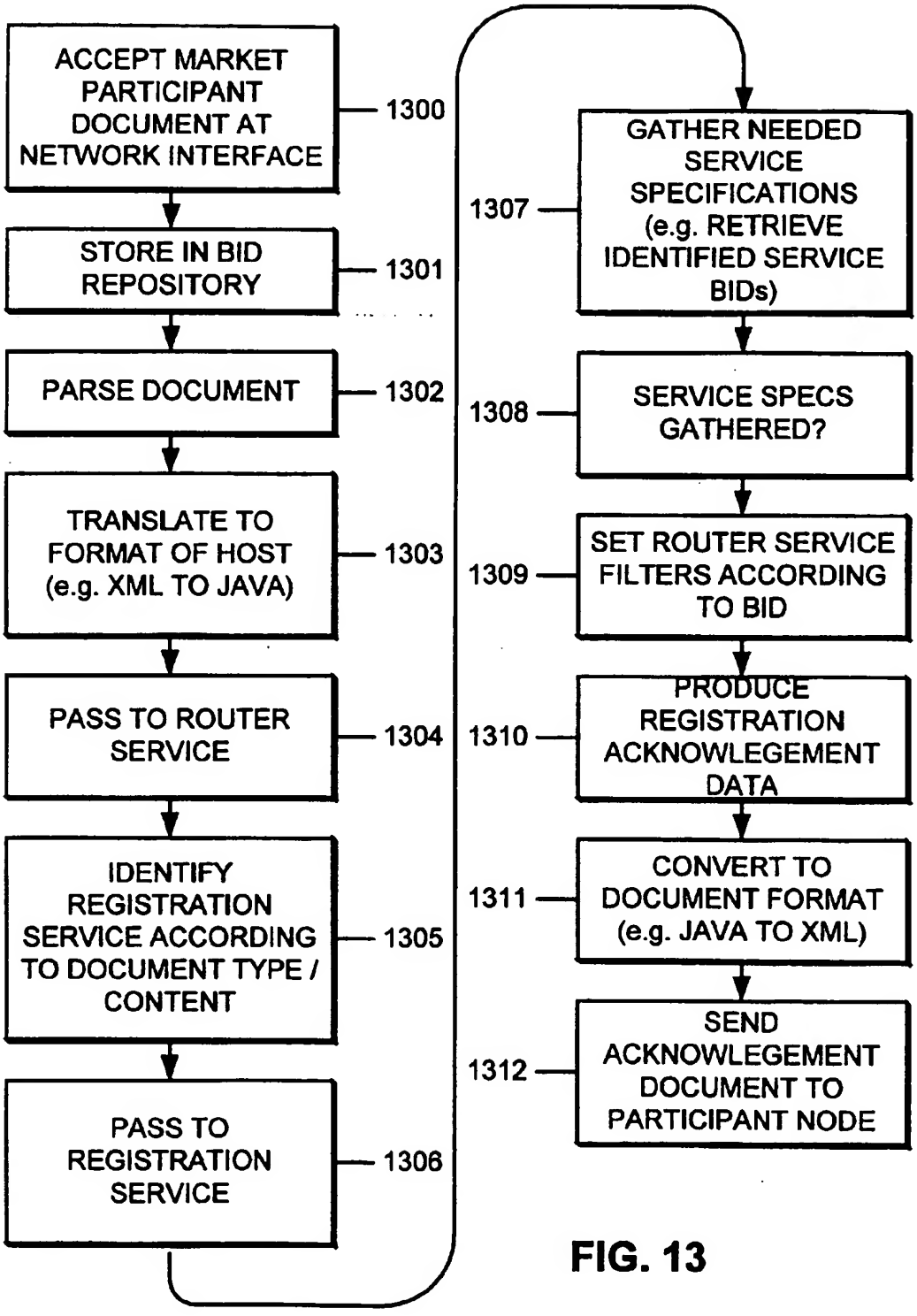

ACCEPT MARKET PARTICIPANT DOCUMENT AT NETWORK INTERFACE
1300
STORE IN BID REPOSITORY
1301
PARSE DOCUMENT
1302
TRANSLATE TO FORMAT OF HOST (e.g. XML TO JAVA)
1303
PASS TO ROUTER SERVICE
1304
IDENTIFY REGISTRATION SERVICE ACCORDING TO DOCUMENT TYPE / CONTENT
1305
PASS TO REGISTRATION SERVICE
1306
GATHER NEEDED SERVICE SPECIFICATIONS (e.g. RETRIEVE IDENTIFIED SERVICE BIDs)
1307
SERVICE SPECS GATHERED?
1308
SET ROUTER SERVICE FILTERS ACCORDING TO BID
1309
PRODUCE REGISTRATION ACKNOWLEGEMENT DATA
1310
CONVERT TO DOCUMENT FORMAT (e.g. JAVA TO XML)
1311
SEND ACKNOWLEGEMENT DOCUMENT TO PARTICIPANT NODE
1312

FIG. 13

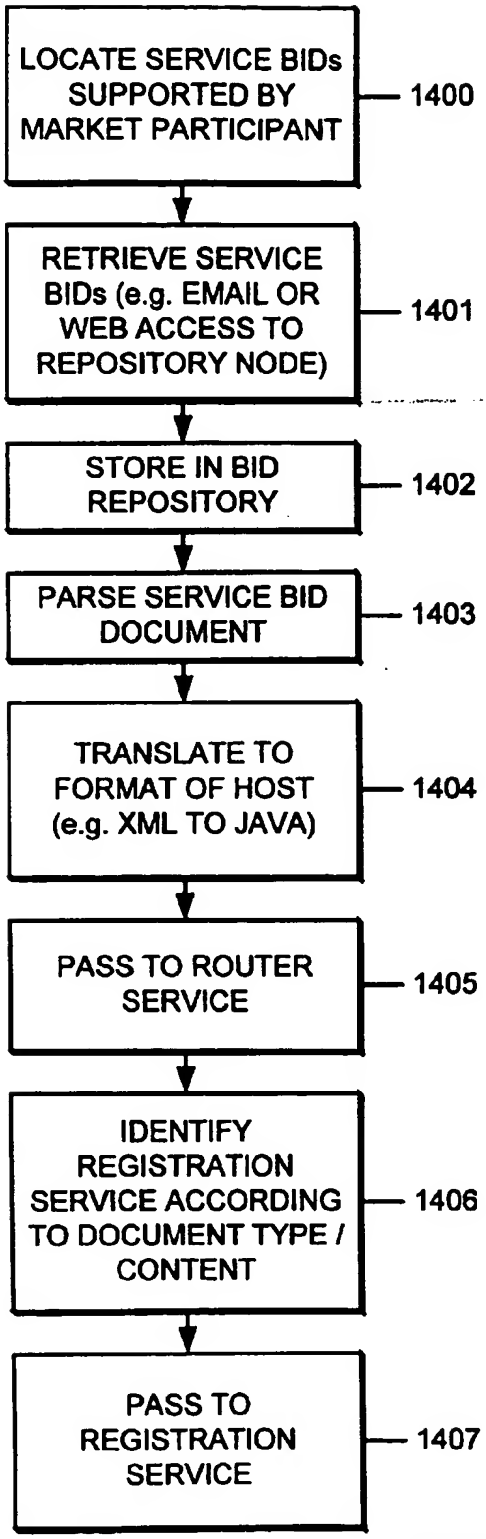

LOCATE SERVICE BIDs SUPPORTED BY MARKET PARTICIPANT
1400
RETRIEVE SERVICE BIDs (e.g. EMAIL OR WEB ACCESS TO REPOSITORY NODE)
1401
STORE IN BID REPOSITORY
1402
PARSE SERVICE BID DOCUMENT
1403
TRANSLATE TO FORMAT OF HOST (e.g. XML TO JAVA)
1404
PASS TO ROUTER SERVICE
1405
IDENTIFY REGISTRATION SERVICE ACCORDING TO DOCUMENT TYPE / CONTENT
1406
PASS TO REGISTRATION SERVICE
1407

FIG. 14

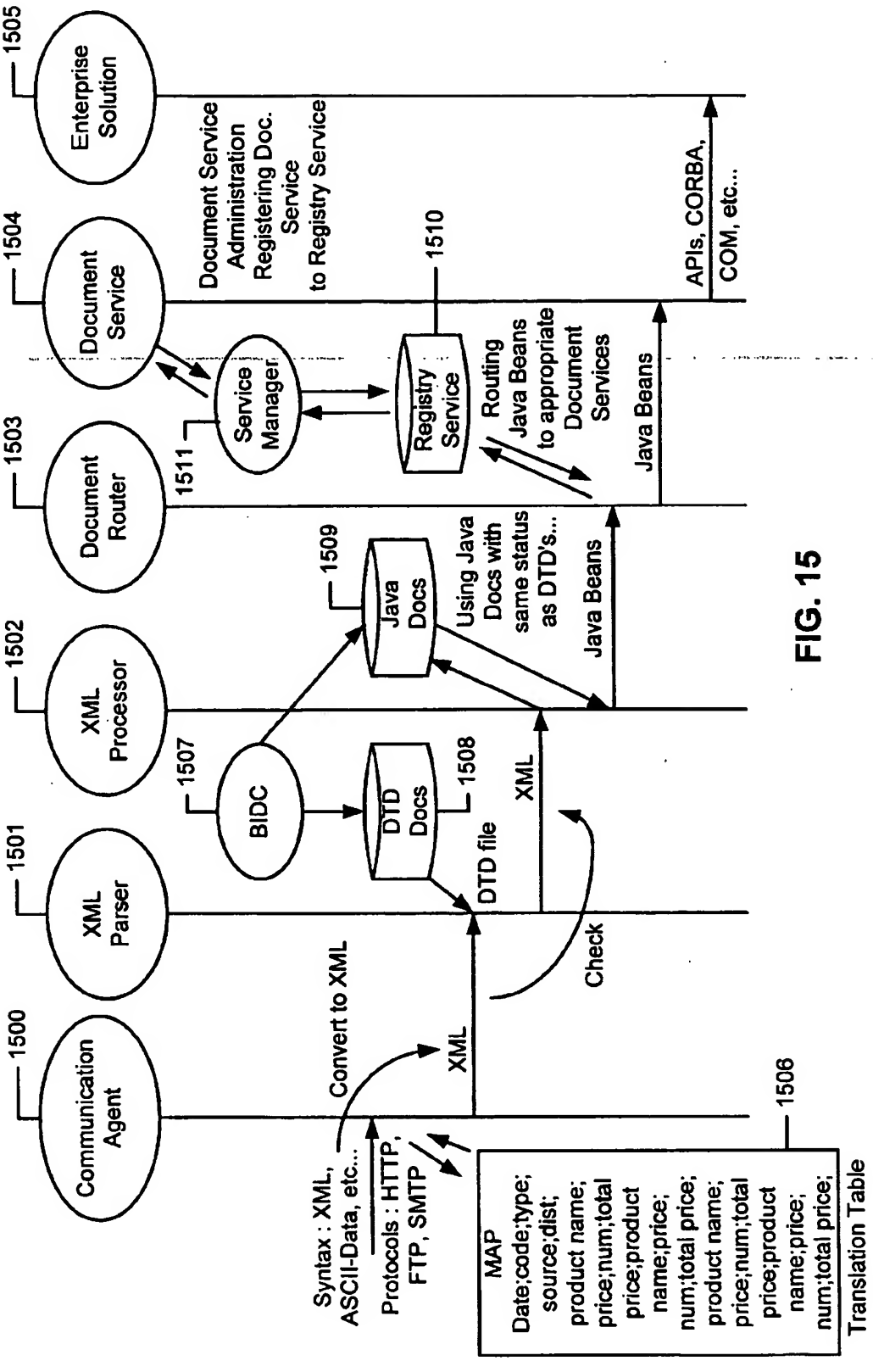

1500
Communication Agent
1501
XML Parser
1502
XML Processor
1503
Document Router
1504
Document Service
1505
Enterprise Solution
1506
Translation Table
1507
BIDC
1508
DTD Docs
1509
Java Docs
1510
Registry Service
1511
Service Manager
Syntax : XML, ASCII-Data, etc...
Protocols : HTTP, FTP, SMTP
Convert to XML
XML
DTD file
Check
XML
Using Java Docs with same status as DTD's...
Java Beans
Routing Java Beans to appropriate Document Services
Java Beans
Document Service Administration Registering Doc. Service to Registry Service
APIs, CORBA, COM, etc...
MAP
Date;code;type;
source;dist;
product name;
price;num;total
price;product
name;price;
num;total price;
product name;
price;num;total
price;product
name;price;
num;total price;

FIG. 15

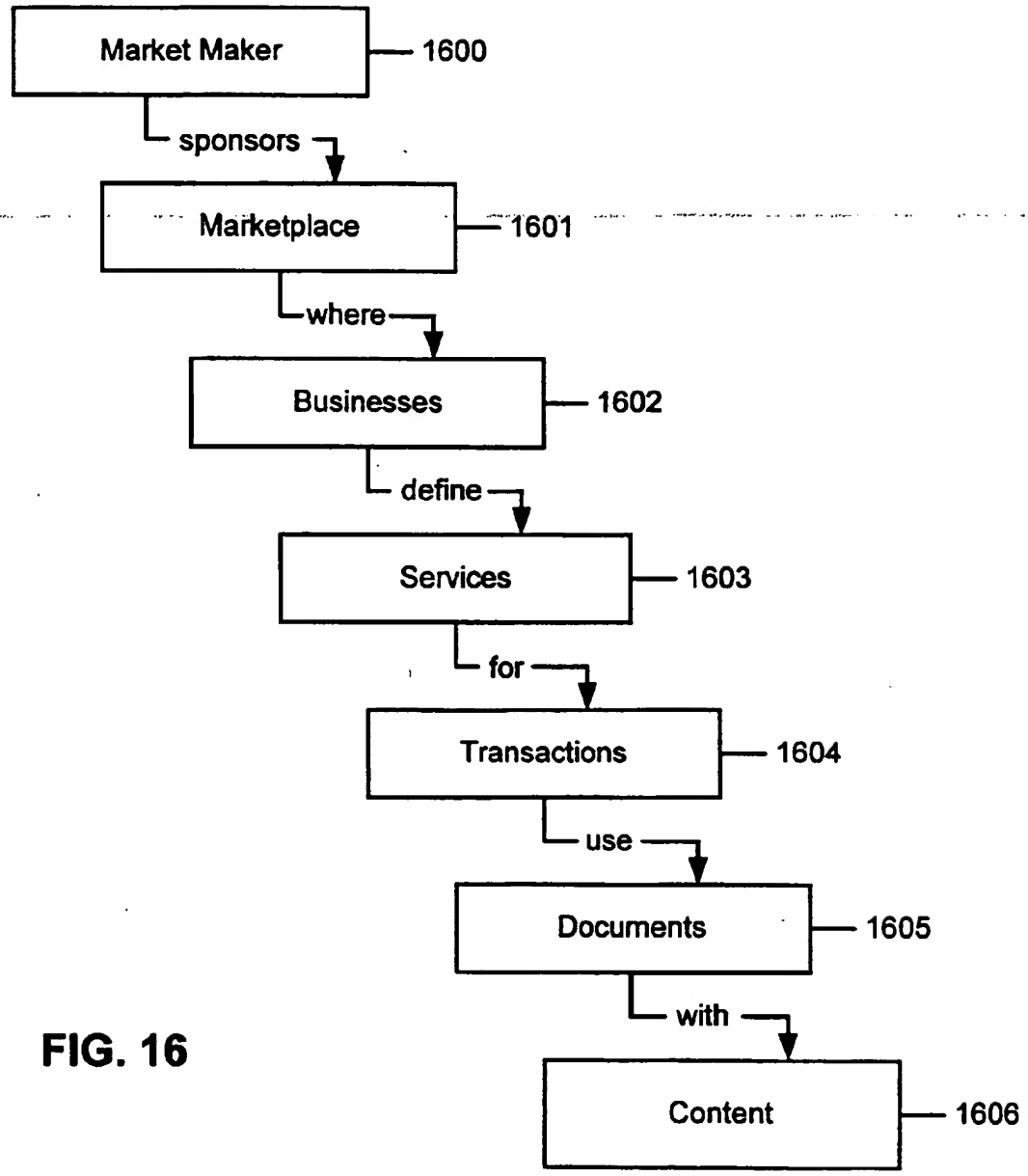

Market Maker
1600
sponsors
Marketplace
1601
where
Businesses
1602
define
Services
1603
for
Transactions
1604
use
Documents
1605
with
Content
1606

FIG. 16

MARKET MAKERS USING DOCUMENTS FOR COMMERCE IN TRADING PARTNER NETWORKS

CROSS-REFERENCE TO RELATED APPLICATIONS

The present application is related to co-pending U.S. patent application Ser. No. 09/173,858, filed on Oct. 16, 1998, the same day as the present application, and having the same inventors, entitled DOCUMENTS FOR COMMERCE IN TRADING PARTNER NETWORKS AND INTERFACE DEFINITIONS BASED ON THE DOCUMENTS; and to co-pending U.S. patent application Ser. No. 09/173,847, filed on Oct. 16, 1998, the same day as the present application, and having the same inventors, entitled PARTICIPANT SERVER WHICH PROCESSES DOCUMENTS FOR COMMERCE IN TRADING PARTNER NETWORKS.

COPYRIGHT NOTICE

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to systems and protocols supporting transactions among diverse clients coupled to a network; and more particularly to systems and protocols supporting commercial transactions among platforms having variant architectures.

2. Description of Related Art

The Internet and other communications networks provide avenues for communication among people and computer platforms which are being used for a wide variety of transactions, including commercial transactions in which participants buy and sell goods and services. Many efforts are underway to facilitate commercial transactions on the Internet. However, with many competing standards, in order to execute a transaction, the parties to the transaction must agree in advance on the protocols to be utilized, and often require custom integration of the platform architectures to support such transactions. Commercial processes internal to a particular node not compatible with agreed upon standards, may require substantial rework for integration with other nodes. Furthermore, as a company commits to one standard or the other, the company becomes locked-in to a given standardized group of transacting parties, to the exclusion of others.

A good overview of the challenges met by Internet commerce development is provided in Tenenbaum, et al., "*Eco System: An Internet Commerce Architecture*", Computer, May 1997, pp. 48–55.

To open commercial transactions on the Internet, standardization of architectural frameworks is desired. Platforms developed to support such commercial frameworks include IBM Commerce Point, Microsoft Internet Commerce Framework, Netscape ONE (Open Network Environment), Oracle NCA (Network Computing Architecture), and Sun/JAVASoft JECF (JAVA Electronic Commerce Framework).

In addition to these proprietary frameworks, programming techniques, such as common distributed object model

based on CORBA IIOP (Common Object Request Broker Architecture Internet ORB Protocol), are being pursued. Use of the common distributed object model is intended to simplify the migration of enterprise systems to systems which can inter-operate at the business application level for electronic commerce. However, a consumer or business using one framework is unable to execute transactions on a different framework. This limits the growth of electronic commerce systems.

Companies implementing one framework will have an application programming interface API which is different than the API's supporting other frameworks. Thus, it is very difficult for companies to access each others business services, without requiring adoption of common business system interfaces. The development of such business system interfaces at the API level requires significant cooperation amongst the parties which is often impractical.

Accordingly, it is desirable to provide a framework which facilitates interaction amongst diverse platforms in a communication network. Such system should facilitate spontaneous commerce between trading partners without custom integration or prior agreement on industry wide standards. Further, such systems should encourage incremental path to business automation, to eliminate much of the time, cost and risks of traditional systems integration.

Overall, it is desirable to provide an electronic commerce system that replaces the closed trading partner networks based on proprietary standards with open markets.

SUMMARY OF THE INVENTION

The present invention offers an infrastructure for connecting businesses with customers, suppliers and trading partners. Under the infrastructure of the present invention, companies exchange information and services using self-defining, machine-readable documents, such as XML (Extensible Markup Language) based documents, that can be easily understood amongst the partners. Documents which describe the documents to be exchanged, called business interface definitions BIDs herein, are posted on the Internet, or otherwise communicated to members of the network. The business interface definitions tell potential trading partners the services the company offers and the documents to use when communicating with such services. Thus, a typical business interface definition allows a customer to place an order by submitting a purchase order, compliant with a document definition published in the BID of a party to receive the purchase order. A supplier is allowed to check availability by downloading an inventory status report compliant with a document definition published in the BID of a business system managing inventory data. Use of predefined, machine-readable business documents provides a more intuitive and flexible way to access enterprise applications.

A node in the commerce network establishes an interface for transactions according to the present invention that comprises a machine-readable specification of an interface, along with a machine-readable data structure that includes interpretation information for the machine-readable specification of the interface. The machine-readable specification of the interface includes a definition of an input document and a definition of an output document, that are accepted and produced by transaction processes for which the node acts as an interface. The definitions of the input and output documents comprise respective descriptions of sets of storage units and logical structures for sets of storage units, such as according to a standard XML based document. The

machine-readable data structure that includes interpretation information according to various aspects of the invention includes data type specifications (e.g. string, array, etc.) for logical structures in the definitions of the input and output documents, content models (e.g. lists of possible values) for logical structures and/or data structures that map predefined sets of storage units for a particular logic structure to respective entries in a list in order to provide a semantic definition of logical structures (e.g. mapping codes to product names).

According to other aspects of the invention, the interface includes a repository in memory accessible by at least one node in the network that stores a library of logic structures and interpretation information for the logic structures. The repository can be extended to include a library of definitions of input and output documents, a library of specifications of interfaces, and a library of specifications for participant interface nodes in the network.

Thus, a participant in the transaction framework of the present invention executes transactions amongst nodes in a network that includes a plurality of nodes executing processes involved in the transactions. The method includes storing a machine-readable specification of an interface for a transaction, the specification includes a definition of an input document and a definition of an output document. The definition of the input and output documents comprise respective descriptions of sets of storage units and logical structures for the sets of storage units. In a preferred system, the definitions are expressed in a manner compliant with a standard XML document type definition DTD, extended by semantic mapping, content modeling and data typing of some elements. The participant in the transaction receives data comprising a document through a communication network. The participant parses the document according to the specification stored for a transaction to identify an input document for the transaction. After parsing the document, at least a portion of the input document is provided in a machine-readable format to a transaction process which produces an output. An output document is formed comprising the output of the transaction process, based on the definition of an output document in the stored specification. The output document is transmitted on the communication network, typically back to the source of the input document, or elsewhere as suits the needs of a particular type of transaction.

Thus the business interface definition bridges the gap between the documents specified for example according to XML and the programs which execute on the front end of the transaction processing services at particular nodes. Such front ends are implemented for example by JAVA virtual machines, or by other common architectures providing for interconnection of systems across a network. The business interface definition provides a technique by which a transaction protocol is programmed using the business interface definition document. The program for the protocol of the transaction is established by a detailed formal specification of a document type.

The machine-readable specification of the interface of the transaction is made accessible to other platforms in the network. Participant platforms include resources to design input documents and output documents according to the transaction interface specified at a complementary node. Thus, participant nodes include resources to access the definition of an input document for the complementary interface and a definition of an output document for the complementary interface. The stored specification for the accessing participant node is established by including at least part of the definition of the output document of the complementary interface in the definition of the input document of the interface stored in the specification.

The process of establishing the stored specification of an interface according to another aspect of the invention includes accessing elements of the machine-readable specification from a repository. The repository stores a library of logic structures, content models, and schematic maps for logic structures, and definition of documents that comprise logic structures used to build interface description. A repository accessible in the network facilitates the building of interface descriptions which can be easily shared. Any differences between the input document created for a particular process and the output document expected as a return by a complementary process can be easily negotiated by communication on the network and agreeing on common logic structures to express particular information.

The machine-readable specification of an interface of a transaction according to one aspect of the invention includes a document compliant with a definition of an interface document that is shared amongst members of the network. A definition of the interface document includes logic structures for storing an identifier of a particular transaction and at least one of definitions and references to definitions of input and output documents for the particular transaction. That is, the interface description for a particular service may actually encompass a definition of the input and output documents. Alternatively, it may include pointers to a location in the repository, or elsewhere in the network, of such definitions.

According to another alternative of the invention, the machine-readable specification includes a business interface definition BID document compliant with a definition of an interface document that includes logical structures for storing an identifier of the interface, and for storing at least one of specifications and references to specifications of a set of one or more transactions supported by the interface. For each supported transaction, the document includes at least one of definitions and references to definitions of input and output documents for the particular transaction.

According to another aspect of the invention, the storage units defined in the definitions of the input and output document comprise parsed data including character data encoding text characters, and mark-up data identifying sets of storage units according to the logical structures of the input and output documents. According to another aspect of the invention, at least one of the sets of storage units encodes the plurality of text characters providing a natural language word. This facilitates the use of the definitions of input and output documents by human readers and developers of such documents.

According to another aspect of the invention, the specification of the input and output documents includes interpretation information for at least one of the sets of storage units identified by the logical structure. The interpretation information in one example encodes definitions for sets of parsed characters. In another example, the interpretation information provides for content model specifications, such as requiring a specific logic structure to carry a member of a list of codes mapped to product descriptions in a catalog. In some systems, the storage units in a logic structure of a document may include sets of unparsed data, as suits the needs of a particular implementation.

According to yet another aspect of the invention, the transaction process in a particular platform has a transaction processing architecture which is one of a plurality of variant

transaction processing architectures. Thus the participant node includes resources for translating at least a portion of the input document into a format readable according to the variant transaction processing architecture of the transaction process utilizing the information. The elements of the document definition are translated into programming objects that include variables and methods according to the variant transaction processing architectures of the transaction process. For a participant having a JAVA virtual machine acting as a transaction process front end, particular fields in the documents are translated into JAVA objects, including the data as well as get and set functions associated with a JAVA object. Other transaction processing architectures supportable according to the present invention include processes compliant with an interface description language in the sense of Common Object Request Broker Architecture CORBA, Component Object Model COM, On-Line Transaction Processing OLTP, and Electronic Data Interchange EDI.

According to other aspects of the invention, a repository is provided that stores document types for use in the plurality of transactions. The machine-readable specification for a particular transaction defines at least one of the input and output documents by reference to a document type in the repository. According to another aspect, the document type included in the repository include a document type for identifying participants in the network. Such document type providing structures for identifying a participant, specifying the services supported by the participant, and specifying the input and output documents for each of such services.

In addition to the methods described above, the present invention provides an apparatus for managing transactions among nodes that includes a network interface, memory for storing data and programs of instructions including a machine-readable specification of an interface for a transaction as described above, and a data processor that is coupled with the memory and the network interface. The programs of instructions stored in the apparatus include logic to execute the processes described above for a participant in the transactions.

The present invention further provides an apparatus for establishing participant interfaces for transactions executed on a system that include a network interface and a data processing resources that execute transaction processes according to a transaction processing architecture. The apparatus includes programs of instructions that are executable by the system and stored on a medium accessible by the system that provide tools to build a definition of a participant interface for a participant in a particular transaction. The definition of a participant interface includes a definition of an input document and a definition of an output document. The definitions of the input and output documents comprise respective machine-readable descriptions of sets of storage units in logical structures for the sets of storage units, which may be compliant in one aspect of the invention with XML document type definitions.

The apparatus for establishing participant interfaces according to this aspect of the invention also includes programs of instructions stored on a medium accessible by the data processing system and responsive to the definitions of input and output documents to compile data structures corresponding to the sets of storage units and logical structures of the input and output documents that are compliant with the transaction processing architecture, to compile instructions executable by the system to translate the input document to the corresponding data structures, and to compile instructions executable by the system to translate output of the transaction processes into sets of storage units and logical structures of the output document.

The tools to build a definition of a participant interface in a preferred system include instructions executable by the system to access elements of the definition from complementary nodes and/or from a repository that stores a library of logical structures and interpretation information for logical structures used to build interface descriptions. According to various aspects of the invention, the repository includes not only a library of logical structures but definitions of documents that comprise logical structures, and formats for specifying participant interfaces. According to this aspect of the invention, tools are provided for building specifications of business interfaces according to the techniques described above in connection with the description of the participant nodes. The tools for building interfaces and the tools for compiling the interfaces into resources needed for communication with transaction processing architectures according to this aspect of the invention, are implemented in participant nodes in the preferred system, and utilized for the development and optimization of the interface descriptions as use of the network grows based on interface descriptions that define input and output documents.

Accordingly, another aspect of the invention provides an apparatus that includes memory and a data processor that executes instructions stored in the memory that include tools to build a definition of a participant interface and a compiler performing the functions described above.

According to yet another aspect of the invention, the use of the participant interface descriptions enables the operation of a market maker node. At such a node, a method for managing transactions is provided that comprises storing machine-readable specifications of a plurality of participant interfaces which identify transactions supported by the interfaces, and the respective input and output documents of such transactions. As described above, the definitions of the input and output documents comprise respective descriptions of sets of storage units and logical structures for the sets of storage units, such as according to the XML standard. In addition, the definitions of the transactions and the definitions of the participant interfaces all comprise documents specified according to a technique compliant with XML or other standardized document expression language. At such market maker node, data comprising a document is received over a communication network. The document is parsed according to the specifications to identify an input document in one or more transactions which accept the identified input document. At least a portion of the input document is provided in a machine-readable format to a transaction process associated with the one or more identified transactions. The step of providing at least a portion of the input document to a transaction process includes executing a routing process according to a processing architecture at the market maker node. The routing process in one aspect of the invention is executed according to a particular processing architecture, and at least a portion of the incoming document is translated into the format of the processing architecture of the routing process. The translating according to the preferred aspect includes producing programming objects that include variables and methods according to the processing architecture of the routing process.

According to another aspect of the invention, the market maker node also supports the repository structure. Thus, a process is included at the market maker node for allowing access by participants in the network to a repository stored at the market maker node. As described above, the repository includes definitions of logic structures, interpretation

information, and document definitions for use by the participant nodes to build transaction interface documents and includes instances of business interface definitions that identify the participants, and the transactions executed by the respective participants.

The routing process includes according to various alternatives the translating of the input document into the variant processing architecture of the processes to which the document is to be routed, or routing the input document in its original document format across the network to a remote processing node, or to combinations of such processes. In alternatives, the routing process may also include techniques for transforming an input document defined according to one input document definition into a different document defined according to a different document specification for a process which has registered to watch for the input document.

Also, the market maker node is provided according to the present invention as apparatus that includes a network interface, memory storing data and programs of instructions including the specifications of the participant interfaces, and a data processor. The logic is provided with the data processor in the form of programs of instructions or otherwise to perform the market maker process as discussed above.

Accordingly, the present invention provides a foundation for electronic commerce based on the sharing of specifications of input and output documents. Documents provide an intuitive and flexible way to access business services, much simpler to implement than programming APIs. It is much easier to interconnect companies in terms of documents that are exchanged, on which such companies already largely agree, than in terms of business system interfaces which invariably differ. In addition, such documents are specified in a human readable format in the preferred embodiment. According to the present invention the business interfaces are specified in documents, such as XML documents that are readily interpretable by humans as well as by computers.

Utilization of the document based transaction architecture of the present invention involves the use of a parser which operates in basically the same way for any source of documents, eliminating much of the need for custom programs to extract and integrate information from each participant in the network. Thus, integrating enterprise information from accounting, purchasing, manufacturing, shipping and other functions can be accomplished by first converting each source to a document having an architecture according to the present invention, and then processing the parsed data stream. Each node in the system that participates in the market only needs to know about the format of its internal systems, as well as the format of the documents being specified for interchange according to the transaction. Thus, there is no need to be able to produce the native format of every other node which might want to participate in the electronic commerce network.

For complete business integration, the present invention provides a repository of standardized logical structures, like XML elements, attributes or meta data, establishing a semantic language for particular commerce communities, a means for mapping between different meta data descriptions, and a server for processing the documents and invoking appropriate applications and services. The basic building blocks of a network according to the present invention include the business interface definitions which tell potential trading partners what online services a company offers and which documents to use to invoke the services; and servers which provide the bridge to bind together the set of internal and external business services to create a trading community. The server operates to parse the incoming documents and invoke the appropriate services. Also the server according to the present invention handles the translation tasks from the format of the documents being received and transmitted, to and from the formats of the respective host systems. Thus, trading partners need only agree on the structure, content and sequencing of the business documents exchanged, and not on the details of application programmer interfaces. How a document is processed and the actions which result from receipt of a document are strictly up to the businesses providing the services. This elevates integration from the system level to the business level. It enables the business to present a clean and stable interface to its partners despite changes in its internal technology implementation, organization or processes.

The whole process of building business interface definitions and enabling servers to manage commerce according to such descriptions is facilitated by a common business library, or repository, of information models for generic business concepts including business description primitives like companies, services and products, business forms like catalogs, purchase orders and invoices, and standard measurements, including time and date, location and classification of goods.

Other aspects of the present invention can be seen upon review of the figures, the detailed description, and the claims which follow.

BRIEF DESCRIPTION OF THE FIGURES

FIG. 1 is a simplified diagram of an electronic commerce network including business interface definitions BIDs according to the present invention.

FIG. 2 is a simplified diagram of a business interface definition document according to the present invention.

FIG. 3 is a conceptual block diagram of a server for a participant node in the network of the present invention.

FIG. 4 is a flow chart illustrating the processing of a received document at a participant node according to the present invention.

FIG. 5 is a block diagram of a parser and transaction process front end for an XML based system.

FIG. 6 is a conceptual diagram of the flow of a parse function.

FIG. 7 is a simplified diagram of the resources at a server used for building a business interface definition according to the present invention.

FIG. 8 is a simplified diagram of a repository according to the present invention for use for building business interface definitions.

FIG. 9 is a flow chart illustrating the processes of building a business interface definition according to the present invention.

FIG. 10 provides a heuristic view of a repository according to the present invention.

FIG. 11 is a simplified diagram of the resources at a server providing the market maker function for the network of the present invention based on business interface definitions.

FIG. 12 is a flow chart for the market maker node processing of a received document.

FIG. 13 is a flow chart illustrating the process of registering participants at a market maker node according to the present invention.

FIG. 14 is a flow chart illustrating the process of providing service specifications at a market maker node according to the process of FIG. 9.

FIG. 15 is a diagram illustrating the sequence of operation at a participant or market maker node according to the present invention.

FIG. 16 is a conceptual diagram of the elements of a commercial network based on BIDs, according to the present invention.

DETAILED DESCRIPTION

A detailed description of the present invention is provided with respect to the figures, in which FIG. 1 illustrates a network of market participants and market makers based on the use of business interface definitions, and supporting the trading of input and output documents specified according to such interface descriptions. The network includes a plurality of nodes 11–18 which are interconnected through a communication network such as the Internet 19, or other telecommunications or data communications network. Each of the nodes 11–19 consists of a computer, such as a portable computer, a desktop personal computer, a workstation, a network of systems, or other data processing resources. The nodes include memory for storing the business interface definition, processors that execute transaction processes supporting commercial transactions with other nodes in the network, and computer programs which are executed by the processors in support of such services. In addition each of the nodes includes a network interface for providing for communication across the Internet 19, or the other communication network.

In the environment of FIG. 1, nodes 11, 12, 13, 14, 16 and 18 are designated market participants. Market participants include resources for consumers or suppliers of goods or services to be traded according to commercial transactions established according to the present invention.

In this example, nodes 15 and 17 are market maker nodes. The market maker nodes include resources for registering business interface definitions, called a BID registry. Participants are able to send documents to a market maker node, at which the document is identified and routed to an appropriate participant which has registered to receive such documents as input. The market maker also facilitates the commercial network by maintaining a repository of standard forms making up a common business library for use in building business interface definitions.

In this example, the market participant 18 is connected directly to the market maker 17, rather than through the Internet 19. This connection directly to the market maker illustrates that the configuration of the networks supporting commercial transactions can be very diverse, incorporating public networks such as the Internet 19, and private connections such as a local area network or a Point-to-Point connection as illustrated between nodes 17 and 18. Actual communication networks are quite diverse and suitable for use to establish commercial transaction networks according to the present invention.

FIG. 2 is a heuristic diagram of nested structures in a business interface definition BID which is established for market participants in the network according to the present invention. The business interface definition illustrated in FIG. 2 is a data structure that consists of logic structures and storage units arranged according to a formal definition of a document structure, such as a XML document type definition DTD. The structure of FIG. 2 includes a first logic structure 200 for identifying a party. Associated with the logic structure 200 are nested logic structures for carrying the name 201, the physical address 202, the network address or location 203, and a set of transactions for a service 204. For each transaction in the service set, an interface definition is provided, including the transaction BID 205, the transaction BID 206, and the transaction BID 207. Within each transaction BID, such as transaction BID 205, logical structures are provided for including a name 208, a location on the network at which the service is performed 209, the operations performed by the service 210, and a set of input documents indicated by the tag 211. Also, the service BID 205 includes a set of output documents indicated by the tag 212. The set of input documents 211 includes a business interface definition for each input document for which the services are designed to respond, including input document business interface definitions 213, 214, and 215. Each business interface definition for an input document includes a name 216, a location on the network at which a description of the document can be found 217, and the modules to be carried in the document as indicated by the field 218. In a similar manner, the output document set 212 includes interface definitions for output documents including the output document BID 219, output document BID 220, and output document BID 221. For each output document BID, a name 222, a location on the network or elsewhere 223, and the modules of the document 224 are specified. The business interface definition for the participant as illustrated in FIG. 2 includes actual definitions of a logic structures to be utilized for the input and output documents of the respective services, or pointers or other references to locations at which these definitions can be found.

In the preferred system, the document of FIG. 2 is specified in an XML document type definition DTD, although other document definition architectures could be used, and includes interpretation information for the logical structures used in interpretation of instances of the documents. In addition, each of the transaction BIDs, input document BIDs and output document BIDs are specified according to an XML document type definitions. The XML type document is an example of a system based on parsed data that includes mark-up data and character data. Mark-up data identifies logical structures within the document and sets of character data identify the content of the logical structures. In addition, unparsed data can be carried in the document for a variety of purposes. See for example the specification of the *Extensible Mark-up Language XML* 1.0 REC-XML-19980210 published by the WC3 XML Working Group at WWW.W3.ORG/TR/1998/REC-XML-19980210.

Thus in an exemplary system, participant nodes in the network establish virtual enterprises by interconnecting business systems and services with XML encoded documents that businesses accept and generate. For example, the business interface definition of a particular service establishes that if a document matching the BID of a request for a catalog entry is received, then a document matching a BID of a catalog entry will be returned. Also, if a document matching the BID of a purchase order is received, and it is acceptable to the receiving terminal, a document matching the BID of an invoice will be returned. The nodes in the network process the XML documents before they enter the local business system, which is established according to the variant transaction processing architecture of any given system in the network. Thus, the system unpacks sets of related documents, such as MIME-encoded sets of XML documents, parses them to create a stream of "mark-up messages". The messages are routed to the appropriate applications and services using for example an event listener model like that described below.

The documents exchanged between business services are encoded using an XML language built from a repository of

building blocks (a common business language) from which more complex document definitions may be created. The repository stores modules of interpretation information that are focused on the functions and information common to business domains, including business description primitives like companies, services and products; business forms like catalogs, purchase orders and invoices; standard measurements, like time, date, location; classification codes and the like providing interpretation information for logical structures in the XML documents.

The business interface definition is a higher level document that acts as a schema used for designing interfaces that trade documents according to the present invention. Thus the business interface definition bridges the gap between the documents specified according to XML and the programs which execute on the front end of the transaction processing services at particular nodes. Such front ends are implemented by JAVA virtual machines, or by other common architectures providing for interconnection of systems across a network. Thus, the business interface definition provides a technique by which a transaction protocol is programmed using the business interface definition document. The program for the protocol of the transaction is established by a detailed formal specification of a document type.

An example business interface definition BID based on a market participant document which conforms to an XML format is provided below. The market participant DTD groups business information about market participants, associating contact and address information with a description of services and financial information. This business information is composed of names, codes, addresses, a dedicated taxonomic mechanism for describing business organization, and a pointer to terms of business. In addition, the services identified by the market participant DTD will specify the input and output documents which that participant is expected respond to and produce. Thus, documents which define schema using an exemplary common business language for a market participant DTD, a service DTD, and a transaction document DTD specified in XML with explanatory comments follow:

Market Participant Sample

```
<!DOCTYPE SCHEMA SYSTBM "bidl.dtd">
<SCHEMA>
<H1 >Market Participant Sample BID</H1>
<META
WHO.OWNS="Veo Systems"          WHO.COPYRIGHT="Veo Systems"
WHEN.COPYRIGHT="1998"           DESCRIPTION="Sample BID"
WHO.CREATED="*"                 WHEN.CREATED="*"
WHAT.VERSION="*"                WHO.MODIFIED="*"
WHEN.MODIFIED="*"               WHEN.EFFECTIVE="*"
WHEN.EXPIRES="*"                WHO.EFFECTIVE="*"
WHO.EXPIRES="*">
</META>
<PROLOG>
<XMLDECL STANDALONE="no"></XMLDECL>
<DOCTYPE NAME="market.participant">
<SYSTBM>markpart.dtd</SYSTEM></DOCTYPE>
</PROLOG>
<DTD NAME="markpart.dtd">
<H2>Market Participant</H2>
<H3>Market Participant</H3>
<ELEMENTTYPE NAME="market.participant">
<EXPLAIN><ITITLE>A Market Participant</ITTLE>
<SYNOPSIS>A business or person and its service interfaces.</SYNOPSIS>
<P>A market participant is a document definition that is created to describe a business and at least
one person with an email address and it presents a set of pointers to service interfaces located on
the network. In this example the pointers have been resolved and the complete BID is presented
here.</P></EXPLAIN>
<MODEL><CHOICE>
<ELEMENT NAME="business"></ELEMENT>
<ELEMENT NAME="person"></ELEMENT>
</CHOICE></MODEL></ELEMENTTYPE>
<H3>Party Prototype</H3>
<PROTOTYPE NAME="party">
<EXPLAIN><ITITLE>The Party Prototype</ITTLE></EXPLAIN>
<MODEL><SEQUENCE>
<ELEMENT NAME="party.name"OCCURS="+"></ELEMENT>
<ELEMENT NAME="address.set"></ELEMENT>
</SEQUENCE></MODEL>
</PROTOTYPE>
<H3>Party Types</H3>
<ELEMENTTYPE NAME="business">
<EXPLAIN><ITITLE>A Business</ITTLE>
<SYNOPSIS>A business (party) with a business number attribute.</SYNOPSIS>
<P>This element inherits the content model of the party prototype and adds a business number
attribute, which serves as a key for database lookup. The business number may be used as a cross-
reference to/from customer id, credit limits contacts lists, etc.</P></EXPLAIN>
<EXTENDS HREF="party">
<ATTDEF NAME="business.number"></REQUIRED></REQUIRED></ATTDBF>
</EXTENDS>
```

-continued

```
</ELEMENTTYPE>
<H3>Person Name</H3>
<ELEMENTTYPE NAME="person">
<EXPLAIN><TITLE>A Person</TITLE></EXPLAIN>
<EXTENDS HREF="party">
<ATTDEF NAME="SSN"><IMPLIED></IMPLIED></ATTDEF>
</EXTENDS>
</ELEMENTTYPE>
<H3>Party Name</H3>
<ELEMENTTYPE NAME="party.name">
<EXPLAIN><TITLE>A Party's Name</TITLE>
<SYNOPSIS>A party's name in a string of character.</SYNOPSIS></EXPLAIN>
<MODEL><STRING></STRING></MODEL>
</ELEMENTTYPE>
<H3>Address Set</H3>
<ELEMENTTYPE NAME="address.set">
<MODEL><SEQUENCE>
<ELEMENT NAME="address.physical"></ELEMENT>
<ELEMENT NAME="telephone"OCCURS="*"></ELEMENT>
<ELEMENT NAME="fax"OCCURS="*"></ELEMENT>
<ELEMENT NAME="email"OCCURS="*"></ELEMENT>
<ELEMENT NAME="internet"OCCURS="*"></ELEMENT>
</SEQUENCE></MODEL>
</ELEMENTTYPE>
<H3>Physical Address</H3>
<ELEMENTTYPE NAME="address.physical">
<EXPLAIN><TITLE>Physical Address</TITLE>
<SYNOPSIS>The street address, city, state, and zip code.</SYNOPSIS></EXPLAIN>
<MODEL><SEQUENCE>
<ELEMENT NAME="street"></ELEMENT>
<ELEMENT NAME="city"></ELEMENT>
<ELEMENT NAME="state"></ELEMENT>
<ELEMENT NAME="postcode"OCCURS="?"></ELEMENT>
<ELEMENT NAME="country"></ELEMENT>
</SEQUENCE></MODEL>
</ELEMENTTYPE>
<H3>Street</H3>
<ELEMENTTYPE NAME="street">
<EXPLAIN><TITLE>Street Address</TITLE>
<SYNOPSIS>Street or postal address.</SYNOPSIS></EXPLAIN>
<MODEL><STRING></STRING></MODEL>
</ELEMENTTYPE>
<H3>City</H3>
<ELEMENTTYPE NAME="city">
<EXPLAIN><TITLE>City Name or Code</TITLE>
<P>The city name or code is a string that contains sufficient information to identity a city within a
designated state.</P>
</EXPLAIN>
<MODEL><STRING></STRING></MODEL>
</ELEMENTTYPE>
<H3>State</H3>
<ELEMENTTYPE NAME="state">
<EXPLAIN><TITLE>State, Province or Prefecture Name or Code</TITLE>
<P>The state name or code contains sufficient information to identfy a state within a designated
country.</P></EXPLAIN>
<MODEL><STRING DATATYPE="COUNTRY.US.SUBENTITY"></STRING></MODEL>
</ELEMENTTYPE>
<H3>Postal Code</H3>
<ELEMENTTYPE NAME="postcode">
<EXPLAIN><TITLE>Postal Code</TITLE>
<P>A postal code is an alphanumeric code, designated by an appropriate postal authority, that is
used to identfy a location or region within the jurisdiction of that postal authority. Postal authorities
include designated national postal authorities.</P></EXPLAIN>
<MODEL><STRING DATATYPE="string"></STRING></MODEL>
</ELEMENTTYPE>
<H3>Country</H3>
<ELEMENTTYPE NAME="country">
<EXPLAIN><TITLE>Country Code</TITLE>
<P>A country code is a two-letter code, designated by ISO, that is used to uniquely identfy a
country.</P></EXPLAIN>
<MODEL><STRING DATATYPE="country"></STRING></MODEL>
</ELEMENTTYPE>
<H3>Network Addresses</H3>
<ELEMENTTYPE NAME="telephone">
<EXPLAIN><TITLE>Telephone Number<TITLE>
<P>A telephone number is a string of alphanumerics and punctuation that uniquely identifies a
telephone service terminal, including extension number.<P></EXPLAIN>
<MODEL><STRING></STRING></MODEL>
</ELEMENTTYPE>
```

-continued

```
<H3>Fax</H3>
<ELEMENTTYPE NAME="fax">
<EXPLAIN><ITITLE>Fax Number</ITITLE>
<P>A fax number is a string of alphanumerics and punctuation that uniquely identifies a fax service
terminal.</P>
</EXPLAIN>
<MODEL><STRING></STRING></MODEL>
</ELEMENTTYPE>
<H3>Email</H3>
<ELEMENTTYPE NAME="email">
<EXPLAIN><ITITLE>Email Address</ITITLE>
<P>An email address is a datatype-constrained string that uniquely identifies a mailbox on a
server.</P></EXPLAIN>
<MODEL><STRING DATATYPE="email"></STRING></MODEL>
</ELEMENTTYPE>
<H3>Internet Address</H3>
<ELEMENTTYPE NAME="internet">
<EXPLAIN><ITITLE>Internet Address</ITITLE>
<P>An Internet address is a datatype-constrained string that uniquely identifies a resource on the
Internet by means of a URL.</P></EXPLAIN>
<MODEL><STRING DATATYPE="url"></STRING></MODEL>
</ELEMENTTYPE>
</DTD>
</SCHEMA>
```

Service Description Sample

```
<!DOCTYPE schema SYSTEM "bidl.dtd">
<SCHEMA>
<H1>Service Description Sample BID</H1>
<META
WHO.OWNS="Veo Systems"          WHO.COPYRIGHT="Veo Systems"
WHEN.COPYRIGHT="1998"           DESCRIPTION="Sample BID"
WHO.CREATED="*"                 WHEN.CREATED="*"
WHAT.VERSION="*"                WHO.MODIFIED="*"
WHEN.MODIFIED="*"               WHEN.EFFECTIVE="*"
WHEN.EXPIRES="*"                WHO.EFFECTIVE="*"
WHO.EXPIRES="*">
</META>
<PROLOG>
<XMLDECL STANDALONE="no"></XMLDECL>
<DOCTYPE NAME="service">
<SYSTEM>service.dtd</SYSTEM></DOCTYPE>
<PROLOG>
<DTD NAME="service.dtd">
<H2>Services</H2>
<H3>Includes</H3>
<!-- INCLUDE><SYSTEM>comments.bim</SYSTEM></INCLUDE-->
<H3>Service Set</H3>
<ELEMENTTYPE NAME="service.set">
<EXPLAIN><ITITLE>Service Set</ITITLE>
<SYNOPSIS>A set of services.</SYNOPSIS></EXPLAIN>
<MODEL>
<ELEMENT NAME="service"OCCURS="+"></ELEMENT>
</MODEL></ELEMENTTYPE>
<H3>Services Prototype</H3>
<PROTOTYPE NAME="prototype.service">
<EXPLAIN><ITITLE>Service</ITITLE></EXPLAIN>
<MODEL><SEQUENCE>
<ELEMENT NAME="service.name"></ELEMENT>
<ELEMENT NAME="service.terms"OCCURS="+"></ELEMENT>
<ELEMENT NAME="service.location"OCCLRS="+"></ELEMENT>
<ELEMENT NAME="service.operation"OCCLRS="+"></ELEMENT>
<SEQUENCE></MODEL>
<!-- ATTGROUP><IMPLEMENTS HREF="common.atrrib"></IMPLEMENTS></ATTGROUP
-->
</PROTOTYPE>
<H3>Service</H3>
<INTRO><P>A service is an addressable network resource that provides interfaces to specific
operations by way of input and output documents.</P></INTRO>
<ELEMENTTYPE NAME="service">
<EXPLAIN><ITITLE>Service</ITITLE>
<P>A service is defined in terms of its name, the location(s) at which the service is available, and
the operation(s) that the service performs.</P></EXPLAIN>
<MODEL><SEQUENCE>
<ELEMENT NAME="service.name"></ELEMENT>
<ELEMENT NAME="service.location"></ELEMENT>
<ELEMENT NAME="service.operation"OCCURS="+"></ELEMENT>
<ELEMENT NAME="service.terms"><ELEMENT>
</SEQUENCE></MODEL>
```

-continued

```
</ELEMENTTYPE>
<H3>Service Name</H3>
<ELEMENTTYPE NAME="service.name">
<EXPLAIN><ITTLE>Service Name</ITTLE>
<P>The service name is a human-readable string that ascribes a moniker for a service. It may be
employed is user interfaces and documentation, or for other purposes.</P></EXPLAIN>
<MODEL><STRING></STRING></MODEL>
</ELEMENTTYPE>
<H3>Service Location</H3>
<ELEMENTTYPE NAME="service.location">
<EXPLAIN><ITTLE>Service Location</ITTLE>
<SYNOPSIS>A LRI of a service.</SYNOPSIS>
<P>A service location is a datatype-constrained string that locates a service on the Internet by
means of a URI.</P></EXPLAIN>
<MODEL><STRING DATATYPE="url"></STRING></MODEL>
</ELEMENTTYPE>
<H3>Service Operations</H3>
<INTRO><P>A service operation consists of a name, location and its interface, as identified by the
type of input document that the service operation accepts and by the type of document that it will
return as a result.</P></INTRO>
<ELEMENTTYPE NAME="service.operation">
<EXPLAIN><ITTLE>Service Operations</ITTLE>
<P>A service operation must have a name, a location, and at least one document type as an input,
with one or more possible document types returned as a result of the operation.</P>
</EXPLAIN>
<MODEL><SEQUENCE>
<ELEMENT NAME="service.operation.name"></ELEMENT>
<ELEMENT NAME="service.operation.location"></ELEMENT>
<ELEMENT NAME="service.operation.input"></ELEMENT>
<ELEMENT NAME="service.operation.output"></ELEMENT>
</SEQUENCE></MODEL>
</ELEMENTTYPE>
<H3>Service Operation Name</H3>
<ELEMENTTYPE NAME="service.operation.name">
<EXPLAIN><ITTLE>Service Operation Name</ITTLE>
<P>The service operation name is a human readable string that ascribes a moniker to a service
operation. It may be employed in user interfaces and documentation, or for other
purposes.</P></EXPLAIN>
<MODEL><STRING></STRING></MODEL>
</ELEMENTTYPE>
<H3>Service Operation Location</H3>
<INTRO><P>The service location is a network resource. That is to say, a URL.</P></INTRO>
<ELEMENTTYPE NAME="service.operation.location">
<EXPLAIN><ITTLE>Service Operation Location"</ITTLE>
<SYNOPSIS>A URI of a service operation.</SYNOPSIS>
<P>A service operation location is a datatype-constrained string that locates a service operation on
the Internet by means of a URL.</P></EXPLAIN>
<MODEL><STRING DATATYPE="url"></STRING></MODEL>
</ELEMENTTYPE>
<H3>Service Operation Input Document</H3>
<INTRO><P>The input to a service operation is defined by its input document type. That is, the
service operation is invoked when the service operation location receives an input document whose
type corresponds to the document type specified by this element.</P>
<P>Rather than define the expected input and output document types in the market participant
document, this example provides pointers to externally-defined BIDs This allows reuse of the same
BID as the input and/or output document type for multiple operations. In addition, it encourages
parallel design and implementation.</P></INTRO>
<ELEMENTTYPE NAME="service.operation.input">
<EXPLAIN><ITTLE>Service Operation Input</ITTLE>
<SYNOPSIS>Identifies the type of the service operation input document.</SYNOPSIS>
<P>Service location input is a datatype-constrained string that identifies a BID on the Internet by
means of a URI.</P>
</EXPLAIN>
<MODEL><STRING DATATYPE="url"></STRING></MODEL>
</ELEMENTTYPE>
<H3>Service Operation Output Document Type</H3>
<INTRO><P>The output of a service operation is defined by its output document type(s). That is,
the service operation is expected to emit a document whose type corresponds to the document type
specified by this element.</P></INTRO>
<ELEMENTTYPE NAME="service.operation.output">
<EXPLAIN><ITTLE>Service Operation Output</ITTLE>
<SYNOPSIS>Identifies the type of the service operation output document.</SYNOPSIS>
<P>Service location output is a datatype-constrained string that identifies a BID on the Internet by
means of a URI.</P>
</EXPLAIN>
<MODEL><STRING DATATYPE="url"></STRING></MODEL>
</ELEMENTTYPE>
<H3>Service Terms</H3>
<INTRO><P>This is a simple collection of string elements, describing the terms of an
```

-continued

```
agreement.</P></INTRO>
<ELEMENTTYPE NAME="service.terms">
<EXPLAIN><TITLE>Service Terms</TITLE>
<SYNOPSIS>Describes the terms of a given agreement.</SYNOPSIS>
</EXPLAIN>
<MODEL><STRING DATATYPE="string"></STRING></MODEL>
</ELEMENTTYPE>
</DTD>
</SCHEMA>
```

The service DTD schema may be extended with a service type element in a common business language repository as follows:

```
<!ELEMENT service.type EMPTY>
<!ATTLIST service.type
        service.type.name (
        catalog.operator
        |commercial.directory.operator
        |eft.services.provider
        |escrower
        |fulfillment.service
        |insurer
        |manufacturer
        |market.operator
        |order.originator
        |ordering.service
        |personal.services.provider
        |retailer
        |retail.aggregator
        |schema.resolution.service
        |shipment.acceptor
        |shipper
        |van
```

The service type element above illustrates interpretation information carried by a business interface definition, in this example a content form allowing identification of any one of a list of valid service types. Other interpretation information includes data typing, such as for example the element <H3> Internet Address </H3> including the content form "url" and expressed in the data type "string." Yet other interpretation information includes mapping of codes to elements of a list, such as for example the element <H3> State </H3> including the code mapping for states in the file "COUNTRY.US-.SUBENTITY."

The service description referred to by the market participant DTD defines the documents that the service accepts and generates upon competition of the service. A basic service description is specified below as a XML document transact.dtd.

Transact.dtd models a transaction description, such as an invoice, or a description of an exchange of value. This document type supports many uses, so the transaction description element has an attribute that allows user to distinguish among invoices, performance, offers to sell, requests for quotes and so on. The exchange may occur among more than two parties, but only two are called out, the offeror and the counter party, each of whom is represented by a pointer to a document conforming to the market participant DTD outlined above. The counter party pointer is optional, to accommodate offers to sell. The exchange description is described in the module tranprim.mod listed below, and includes pricing and subtotals. Following the exchange description, charges applying to the transaction as a whole may be provided, and a total charge must be supplied. Thus, the transaction description schema document transact.dtd for this example is set forth below:

```
<!-- transact.dtd Version: 1.0 -->
<!-- Copyright 1998 Veo Systems, Inc. -->
. . .
<!ELEMENT transaction.description (meta?, issuer.pointer,
        counterparty.pointer?, exhange.descrption+, general.charges?,
        net.total?)>
<!ATTLIST transaction.description
        transaction.type (invoice | pro.forma | offer.to.sell | order
                |request.for.quote | request.for.bid
                |request.for.proposal | response.to.request.for.quote
                |response.to.requestfor.bid
                |response.to.request.for.proposal) "invoice"
        %common.attrib;
        %altrep.attrib;
        %ttl.attrib;
```

Representative market participant, and service DTDs, created according to the definitions above, are as follows:

Market Participant DTD

```
<!ELEMENT business (party.name+, address.set)>
<!ATTLIST business business.number CDATA #REQUIRED
>
<!ELEMENT party.name (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT internet (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT address.physical (street, city, state postcode?, country)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT person (party.name+, address.set)>
<!ATTLIST person SSN CDATA #IMPLIED
>
```

-continued

```
<!ELEMENT fax (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT address.set (address.physical, telephone*, fax*, email*, internet*)>
<!ELEMENT postcode (#PCDATA)>
<!ELEMENT market.participant (business | person)>
```

Service DTD

```
<!ELEMENT service.location (#PCDATA)>
<!ELEMENT service.terms (#PCDATA)>
<!ELEMENT service.operation.name (#PCDATA)>
<!ELEMENT service.operation (service.operation.name, service.operation.location,
service.operation.input, service.operation. output)>
<!ELEMENT service (service.name, service.location, service.operation+, service.terms) >
<!ELEMENT service.operation.input (#PCDATA)>
<!ELEMENT service.operation.location (#PCDATA)>
<!ELEMENT service.name (#PCDATA)>
<!ELEMENT service.set (service+)>
<!ELEMENT service.operation.output (#PCDATA)>
```

One instance of a document produced according to the transact.dtd follows:

```
<?xml version="1.0"?>
<!--rorder.xml Version: 1.0 -->
<!Copyright 1998 Veo Systems, Inc. -->
<!DOCTYPE transaction.description SYSTEM "urn:x-veosystems:dtd:cbl:transact: 1.0:>
<transaction.description transaction.type="order">
<meta>
<urn?urn:x-veosystems:doc:00023
</urn>
        <thread.id party.assigned.by="reqorg">FRT876
        </thread.id>
</meta>
        <issuer.pointer>
            <xll.locator urllink="reqorg.xml">Customer
            Pointer
            </xll.locator>
        </issuer.pointer>
        <counterparty.pointer>
            <xll.locator urllink="compu.xml">Catalog entry owner
            pointer
            </xll.locator>
        </counterparty.pointer>
<exchange.description>
<line.item>
        <product.instance>
        <product.description.pointer>
            <xll.locator urllink="cthink.xml">Catalogue Entry      Pointer
            <xll.locator>
        </product.description.pointer>
        <product.specifics>
        <info.description.set>
<info.description>
<xml.descriptor>
        <doctype>
        <dtd system.id="urn:x-veosystems:dtd:cbl:gprod:1.0"/>
        <doctype>
        <xml.descriptor.details>
        <xll.xptr.frag>DESCENDANT(ALL, os)STRING("Windows
95")
        </xll.xptr.frag>
        </xll.xptr.frag>DECENDANT(ALL,p.speed)STRING("200")
        </xll.xptr.frag>
        </xll.xptr.frag>DESCENDANT(ALL,hard.disk.capacity)
        STRING("4")
        </xll.xptr.frag>
        </xll.xptr.frag>DESCENDANT(ALL,d.size)STRING("14.1")
        </xll.xptr.frag>
        </xml.descriptor.details>
</xml.descriptor>
</info.description>
        </info.description.set>
        </product.specifics>
```

-continued

```
</quantity>1
</quantity>
        </product.instance>
<shipment.coordinates.set>
<shipment.coordinates>
<shipment.destination>
            <address.set>
        <address.named>SW-1
        <address.named>
        <address.physical>
            <building.sublocation>208C</building.sublocation>
            <location.in.street>123
            </location.in. street>
            <street>Frontage Rd.
            </street>
            <city>Beltway
            </city>
            <country.subentity.us
            county.subentity.us.name="MD"/>
            <postcode>20000
            <postcode>
        </address.physical>
        <telephone>
            <telephone.number>617-666-2000
            </telephone.number>
            <telephone.extension>1201
            </telephone.extension>
        </telephone>
        </address.set>
</shipment.destination>
</shipment.special>No deliveries after 4 PM</shipment.special>
</shipment.coordinates>
</shipment.coordinates.set>
        <payment.set>
        <credit.card
        issuer.name="VISA"
        instrument.number="3787-812345-67893"
        expiry.date="12/97"
        currency.code="USD">
        <amount.group>
            <amount.monetary currency.code="USD">3975
            </amount.monetary>
        </amount.group>
        </payment.set>
</line.item>
</exhange.description>
</tansaction.description>
```

Accordingly, the present invention provides a technique by which a market participant is able to identify itself, and identify the types of input documents and the types of output documents with which it is willing to transact business. The particular manner in which the content carried in such documents is processed by the other parties to the transaction, or by the local party, is not involved in establishing a business relationship nor carrying out transactions.

FIG. **3** provides a simplified view of a participant node in the network according to the present invention. The node illustrated in FIG. **3** includes a network interface **300** which is coupled to a communication network on port **301**. The network interface is coupled to a document parser **301**. The parser **301** supplies the logical structures from an incoming document to a translator module **302**, which provides for translating the incoming document into a form usable by the host transaction system, and vice versa translating the output of host processes into the format of a document which matches the output document form in the business interface definition for transmission to a destination. The parser **301** and translator **302** are responsive to the business interface definition stored in the participant module **303**.

The output data structures from the translator **302** are supplied to a transaction process front end **304** along with events signaled by the parser **301**. The front end **304** in one embodiment consists of a JAVA virtual machine or other similar interface adapted for communication amongst diverse nodes in a network. The transaction processing front end **304** responds to the events indicated by the parser **301** and the translator **302** to route the incoming data to appropriate functions in the enterprise systems and networks to which the participant is coupled. Thus, the transaction process front end **304** in the example of FIG. **3** is coupled to commercial functions **305**, database functions **306**, other enterprise functions such as accounting and billing **307**, and to the specific event listeners and processors **308** which are designed to respond to the events indicated by the parser.

The parser **301** takes a purchase order like that in the example above, or other document, specified according to the business interface definition and creates a set of events that are recognized by the local transaction processing architecture, such as a set of JAVA events for a JAVA virtual machine.

The parser of the present invention is uncoupled from the programs that listen for events based on the received documents. Various pieces of mark-up in a received document or a complete document meeting certain specifications serve as instructions for listening functions to start processing. Thus listening programs carry out the business logic associated with the document information. For example, a program associated with an address element may be code that validates the postal code by checking the database. These

listeners subscribe to events by registering with a document router, which directs the relevant events to all subscribers who are interested in them.

For example, the purchase order specified above may be monitored by programs listening for events generated by the parser, which would connect the document or its contents to an order entry program. Receipt of product descriptions within the purchase order, might invoke a program to check inventory. Receipt of address information within the purchase order, would then invoke a program to check availability of services for delivery. Buyer information fields in the document, could invoke processes to check order history for credit worthiness or to offer a promotion or similar processing based on knowing the identity of the consumer.

Complex listeners can be created as configurations of primitive ones. For example, a purchase order listener may contain and invoke the list listeners set out in the previous paragraph, or the list members may be invoked on their own. Note that the applications that the listeners run are unlikely to be native XML processes or native JAVA processes. In these cases, the objects would be transformed into the format required by the receiving trans application. When the application finishes processing, its output is then transformed back to the XML format for communication to other nodes in the network.

It can be seen that the market participant document type description, and the transaction document type description outlined above include a schematic mapping for logic elements in the documents, and include mark-up language based on natural language. The natural language mark-up, and other natural language attributes of XML facilitate the use of XML type mark-up languages for the specification of business interface definitions, service descriptions, and the descriptions of input and output documents.

The participant module **303** in addition to storing the business interface definition includes a compiler which is used to compile objects or other data structures to be used by the transaction process front end **304** which corresponds to the logical structures in the incoming documents, and to compile the translator **302**. Thus, as the business interface definition is modified or updated by the participant as the transactions with which the participant is involved change, the translator **302** and parser **301** are automatically kept up to date.

In a preferred system, the set of JAVA events is created by a compiler which corresponds to the grove model of SGML, mainly the standard Element Structure Information Set augmented by the "property set" for each element. *International Standard ISO/IEC* 10179:1996 (*E*), *Information Technology—Processing Languages—Document Style Semantics and Specification Language* (*DSSSL*). Turning the XML document into a set of events for the world to process contrasts with the normal model of parsing in which the parser output is maintained as an internal data structure. By translating the elements of the XML document into JAVA events or other programming structures that are suitable for use by the transaction processing front end of the respective nodes enables rich functionality at nodes utilizing the documents being traded.

Thus, the transaction process front end **304** is able to operate in a publish and subscribe architecture that enables the addition of new listener programs without the knowledge of or impact on other listening programs in the system. Each listener, **305, 306, 307, 308** in FIG. **3**, maintains a queue in which the front end **304** directs events. This enables multiple listeners to handle events in parallel at their own pace.

Furthermore, according to the present invention the applications that the listeners run need not be native XML functions, or native functions which match the format of the incoming document. Rather, these listeners may be JAVA functions, if the transaction process front end **304** is a JAVA interface, or may be functions which run according to a unique transaction processing architecture. In these cases, the objects would be transformed into the format required by the receiving application. When the application of the listener finishes, its output is then transformed back into the format of a document as specified by the business interface definition in the module **303**. Thus, the translator **302** is coupled to the network interface **300** directly for supplying the composed documents as outputs.

The listeners coupled to the transaction processing front end may include listeners for input documents, listeners for specific elements of the input documents, and listeners for attributes stored in particular elements of the input document. This enables diverse and flexible implementations of transaction processes at the participant nodes for filtering and responding to incoming documents.

FIG. **4** illustrates a process of receiving and processing an incoming document for the system of FIG. **3**. Thus, the process begins by receiving a document at the network interface (step **400**). The parser identifies the document type (**401**) in response to the business interface definition. Using the business interface definition, which stores a DTD for the document in the XML format, the document is parsed (step **402**). Next, the elements and attributes of the document are translated into the format of the host (step **403**). In this example, the XML logic structures are translated into JAVA objects which carry the data of the XML element as well as methods associated with the data such as get and set functions. Next, the host objects are transferred to the host transaction processing front end (step **404**). These objects are routed to processes in response to the events indicated by the parser and the translator. The processes which receive the elements of the document are executed and produce an output (step **405**). The output is translated to the format of an output document as defined by the business interface definition (step **406**). In this example, the translation proceeds from the form of a JAVA object to that of an XML document. Finally, the output document is transmitted to its destination through the network interface (step **407**).

FIG. **5** is a more detailed diagram of the event generator/event listener mechanism for the system of FIG. **3**. In general the approach illustrated in FIG. **5** is a refinement of the JAVA JDK 1.1 event model. In this model, three kinds of objects are considered. A first kind of object is an event object which contains information about the occurrence of an event. There may be any number of kinds of event objects, corresponding to all the different kinds of events which can occur. A second kind of object is an Event generator, which monitors activity and generates event objects when something happens. Third, event listeners, listen for event objects generated by event generators. Event listeners generally listen to specific event generators, such as for mouse clicks on a particular window. Event listeners call an "ADD event listener" method on the event generator. This model can be adapted to the environment of FIG. **3** in which the objects are generated in response to parsing and walking a graph of objects, such as represented by an XML document.

The system illustrated in FIG. **5** includes a generic XML parser **500**. Such parser can be implemented using a standard call back model. When a parsing event occurs, the parser calls a particular method in an application object, passing in the appropriate information in the parameters. Thus a single application **501** resides with the parser. The application packages the information provided by the parser in an XML event object and sends it to as many event listeners as have identified themselves, as indicated by the block **502**. The set of events **502** is completely parser independent. The events

502 can be supplied to any number of listeners and any number of threads on any number of machines. The events are based on the element structure information set ESIS in one alternative. Thus, they consist of a list of the important aspects of a document structure, such as the starts and ends of elements, or of the recognition of an attribute. XML (and SGML) parsers generally use the ESIS structure as a default set of information for a parser to return to its application.

A specialized ESIS listener **503** is coupled to the set of events **502**. This listener **503** implements the ESIS listener API, and listens for all XML events from one or more generators. An element event generator **504** is a specialized ESIS listener which is also an XML event generator. Its listeners are objects only interested in events for particular types of elements. For example in an HTML environment, the listener may only be interested in ordered lists, that is only the part of the document between the <OL> and </OL> tags. For another example, a listener may listen for "party.name" elements, or for "service.name" elements according to the common business language, from the example documents above, process the events to ensure that the elements carry data that matches the schematic mapping for the element, and react according to the process needed at the receiving node.

This allows the system to have small objects that listen for particular parts of the document, such as one which only adds up prices. Since listeners can both add and remove themselves from a generator, there can be a listener which only listens to for example the <HEAD> part of an HTML document. Because of this and because of the highly recursive nature of XML documents, it is possible to write highly targeted code, and to write concurrent listeners. For example, an <OL> listener can set up an <LI> listener completely separate from the manner in which the <UL> (unordered list) listener sets up its <LI> listener. Alternatively, it can create a listener which generates a graphic user interface and another which searches a database using the same input. Thus, the document is treated as a program executed by the listeners, as opposed to the finished data structure which the application examines one piece at a time. If an application is written this way, it is not necessary to have the entire document in memory to execute an application.

The next listener coupled to the set of events **502** is an attribute filter **505**. The attribute filter **505** like the element filter **504** is an attribute event generator according to the ESIS listener model. The listener for an attribute filter specifies the attributes it is interested in, and receives events for any element having that attribute specified. So for example, a font manager might receive events only for elements having a font attribute, such as the <P FONT= "Times Roman" /P>.

The element event generator **504** supplies such element objects to specialize the element listeners **504A**.

The attribute event generator **505** supplies the attribute event objects to attribute listeners **505A**. Similarly, the attribute objects are supplied to a "architecture" in the sense of an SGML/XML transformation from one document type to another using attributes. Thus the architecture of **505B** allows a particular attribute with a particular name to be distinguished. Only elements with that attribute defined become part of the output document, and the name of the element in the output document is the value of the attribute in the input document. For example, if the architecture **505B** is HTML, the string:

<PURCHASES HTML="OL"><ITEM HTML= "LI"><NAME HTML="B">STUFF</ NAME><PRICE HTML="B">123</PRICE></ ITEM></PURCHASES>

translates into:

<OL><LI><B>STUFF</B><B>123</B></LI></OL>

which is correct HTML.

The next module which is coupled to the set of events **502** is a tree builder **506**. The tree builder takes a stream of XML events and generates a tree representation of the underlying document. One preferred version of the tree builder **506** generates a document object model DOM object **507**, according to the specification of the W3C (See, http://www.w3.org/TR/1998/WD-DOM-19980720/introduction.html). However listeners in event streams can be used to handle most requirements, a tree version is useful for supporting queries around a document, reordering of nodes, creation of new documents, and supporting a data structure in memory from which the same event stream can be generated multiple times, for example like parsing the document many times. A specialized builder **508** can be coupled to the tree builder **506** in order to build special subtrees for parts of the document as suits a particular implementation.

In addition to responses to incoming documents, other sources of XML events **502** can be provided. Thus, an event stream **510** is generated by walking over a tree of DOM objects and regenerating the original event stream created when the document was being parsed. This allows the system to present the appearance that the document is being parsed several times.

The idea of an object which walks a tree and generates a stream of events can be generalize beyond the tree of DOM objects, to any tree of objects which can be queried. Thus, a JAVA walker **512** may be an application which walks a tree of JAVA bean components **513**. The walker walks over all the publicly accessible fields and methods. The walker keeps track of the objects it has already visited to ensure that it doesn't go into an endless cycle. JAVA events **514** are the type of events generated by the JAVA walker **512**. This currently includes most of the kinds of information one can derive from an object. This is the JAVA equivalent of ESIS and allows the same programming approach applied to XML to be applied to JAVA objects generally, although particularly to JAVA beans.

The JAVA to XML event generator **515** constitutes a JAVA listener and a JAVA event generator. It receives the stream of events **514** from the JAVA walker **512** and translates selected ones to present a JAVA object as an XML document. In the one preferred embodiment, the event generator **515** exploits the JAVA beans API. Each object seen becomes an element, with the element name the same as the class name. Within that element, each embedded method also becomes an element whose content is the value returned by invoking the method. If it is an object or an array of objects, then these are walked in turn.

FIG. **6** outlines a particular application built on the framework of FIG. **5**. This application takes in an XML document **600** and applies it to a parser/generator **601**. ESIS events **602** are generated and supplied to an attribute generator **603** and tree builder **604**. The attribute generator corresponds to the generator **505** of FIG. **5**. It supplies the events to the "architecture" **505B** for translating the XML input to an HTML output for example. These events are processed in parallel as indicated by block **605** and processed by listeners. The output of the listeners are supplied to a document writer **506** and then translated back to an XML format for output. Thus for example this application illustrated in FIG. **6** takes an XML document and outputs an HTML document containing a form. The form is then sent

to a browser, and the result is converted back to XML. For this exercise, the architecture concept provides the mapping from XML to HTML. The three architectures included in FIG. **6** include one for providing the structure of the HTML document, such as tables and lists, a second specifying text to be displayed, such as labels for input fields on the browser document, and the third describes the input fields themselves. The elements of the XML document required to maintain the XML documents structure become invisible fields in the HTML form. This is useful for use in reconstruction of the XML document from the information the client will put into the HTTP post message that is sent back to the server. Each architecture takes the input document and transforms it into an architecture based on a subset of HTML. Listeners listening for these events, output events for the HTML document, which then go to a document writer object. The document writer object listens to XML events and turns them back into an XML document. The document writer object is a listener to all the element generators listening to the architectures in this example.

The organization of the processing module illustrated in FIGS. **5** and **6** is representative of one embodiment of the parser and transaction process front end for the system of FIG. **3**. As can be seen, a very flexible interface is provided by which diverse transaction processes can be executed in response to the incoming XML documents, or other structured document formats.

FIG. **7** illustrates a node similar to that of FIG. **3**, except that it includes a business interface definition builder module **700**. Thus, the system of FIG. **7** includes a network interface **701**, a document parser **702**, and a document translator **703**. The translator **703** supplies its output to a transaction processing front end **704**, which in turn is coupled to listening functions such as commercial functions **705**, a database **706**, enterprise functions **707**, and other generic listeners and processors **708**. As illustrated in FIG. **7**, the business interface definition builder **700** includes a user interface, a common business library CBL repository, a process for reading complementary business interface definitions, and a compiler. The user interface is used to assist an enterprise in the building of a business interface definition relying on the common business library repository, and the ability to read complementary business interface definitions. Thus, the input document of a complementary business interface definition can be specified as the output document of a particular transaction, and the output document of the complementary business interface definition can be specified as the input to such transaction process. In a similar manner a transaction business interface definition can be composed using components selected from the CBL repository. The use of the CBL repository encourages the use of standardized document formats, such as the example schema (bid1) documents above, logical structures and interpretation information in the building of business interface definitions which can be readily adopted by other people in the network.

The business interface definition builder module **700** also includes a compiler which is used for generating the translator **703**, the objects to be produced by the translator according to the host transaction processing architecture, and to manage the parsing function **702**.

FIG. **8** is a heuristic diagram showing logical structures stored in the repository in the business interface definition builder **700**. Thus, the repository storage representative party business interface definitions **800**, including for example a consumer BID **801**, a catalog house BID **802**, a warehouse BID **803**, and an auction house BID **804**. Thus, a new participant in an online market may select as a basic interface description one of the standardized BIDs which best matches its business. In addition, the repository will store a set of service business interface definitions **805**. For example, an order entry BID **806**, an order tracking BID **807**, an order fulfillment BID **808**, and a catalog service BID **809** could be stored. As a new participant in the market builds a business interface definition, it may select the business interface definitions of standardized services stored in the repository.

In addition to the party and service BIDs, input and output document BIDs are stored in the repository as indicated by the field **810**. Thus, a purchase order BID **811**, an invoice BID **812**, a request for quote BID **813**, a product availability report BID **814**, and an order status BID **815** might be stored in the repository.

The repository, in addition to the business interface definitions which in a preferred system are specified as document type definitions according to XML, stores interpretation information in the form of semantic maps as indicated by the field **816**. Thus, semantic maps which are used for specifying weights **817**, currencies **818**, sizes **819**, product identifiers **820**, and product features **821** in this example might be stored in the repository. Further, the interpretation information provides for typing of data structures within the logical structures of documents.

In addition, logical structures used in the composing of business interface definitions could be stored in the repository as indicated by block **822**. Thus, forms for providing address information **823**, forms for providing pricing information **824**, and forms for providing terms of contractual relationships could be provided **825**. As the network expands, the CBL repository will also expand and standardize tending to make the addition of new participants, and the modification of business interface definitions easier.

FIG. **9** illustrates the process of building a business interface definition using the system of FIG. **7**. The process begins by displaying a BID builder graphical interface to the user (step **900**). The system accepts user input identifying a participant, service and document information generated by the graphical interface (step **901**).

Next, any referenced logical structures, interpretation information, document definitions and/or service definitions are retrieved from the repository in response to user input via the graphical user interface (step **902**). In the next step, any complementary business interface definitions or components of business interface definitions are accessed from other participants in the network selected via user input, by customized search engines, web browsers or otherwise (step **903**). A document definition for the participant is created using the information gathered (step **904**). The translators for the document to host and host to document mappers are created by the compiler (step **905**). Host architecture data structures corresponding to the definition are created by the compiler (step **906**), and the business interface definition which has been created is posted on the network, such as by posting on a website or otherwise, making it accessible to other nodes in the network (step **907**).

Business interface definitions tell potential trading partners the online services the company offers and which documents to use to invoke those services. Thus, the services are defined in the business interface definition by the documents that they accept and produce. This is illustrated in the following fragment of an XML service definition.

```
<service>
<service.name>Order Service</service.name>
<service.location>www.veosystems.com/order</service.location>
<service.op>
<service.op.name>Submit Order</service.op.name>
<service.op.inputdoc>www.commerce.net/po.dtd</service.op.inputdoc>
<service.op.outputdoc>
```

-continued

```
        www.veosystems.com/invoice.dtd</service.op.outputdoc>
</service.op>
<service.op>
<service.op.name>Track Order</service.op.name>
<service.op.inputdoc>www.commerce.net
        /request.track.dtd<service.op.inputdoc>
<service.op.outputdoc>
        www.veosystems.com/response.track.dtd<service.op.outputdoc>
</service.op>
</service>
```

This XML fragment defines a service consisting of two transactions, one for taking orders and the other for tracking them. Each definition expresses a contract or promise to carry out a service if a valid request is submitted to the specified Web address. The Order service here requires an input document that conforms to a standard "po.dtd" Document Type Definition located in the repository, which may be local, or stored in an industry wide registry on the network. If a node can fulfill the order, it will return a document conforming to a customized "invoice.dtd" whose definition is local. In effect, the company is promising to do business with anyone who can submit a Purchase Order that conforms to the XML specification it declares. No prior arrangement is necessary.

The DTD is the formal specification or grammar for documents of a given type; it describes the elements, their attributes, and the order in which they must appear. For example, purchase orders typically contain the names and addresses of the buyer and seller, a set of product descriptions, and associated terms and conditions such as price and delivery dates. In Electronic Data Interchange EDI for example, the X12 850 specification is a commonly used model for purchase orders.

The repository encourages the development of XML document models from reusable semantic components that are common to many business domains. Such documents can be understood from their common message elements, even though they may appear quite different. This is the role of the Common Business Library repository.

The Common Business Library repository consists of information models for generic business concepts including:

- business description primitives like companies, services, and products;
- business forms like catalogs, purchase orders, and invoices;
- standard measurements, date and time, location, classification codes.

This information is represented as an extensible, public set of XML building blocks that companies can customize and assemble to develop XML applications quickly. Atomic CBL elements implement industry messaging standards and conventions such as standard ISO codes for countries, currencies, addresses, and time. Low level CBL semantics have also come from analysis of proposed metadata frameworks for Internet resources, such as Dublin Core.

The next level of elements use these building blocks to implement the basic business forms such as those used in X12 EDI transactions as well as those used in emerging Internet standards such as OTP (Open Trading Protocol) and OBI (Open Buying on the Internet).

CBL's focus is on the functions and information that are common to all business domains (business description primitives like companies, services, and products; business forms like catalogs, purchase orders, and invoices; standard measurements, date and time, location, classification codes). CBL builds on standards or industry conventions for semantics where possible (e.g., the rules that specify "day/month/year" in Europe vs "month/day/year" in the U.S. are encoded in separate CBL modules).

The CBL is a language that is used for designing applications. It is designed to bridge the gap between the "document world" of XML and the "programming world" of JAVA or other transaction processing architectures. Schema embodies a philosophy of "programming with documents" in which a detailed formal specification of a document type is the master source from which a variety of related forms can be generated. These forms include XML DTDs for CBL, JAVA objects, programs for converting XML instances to and from the corresponding JAVA objects, and supporting documentation.

The CBL creates a single source from which almost all of the pieces of a system can be automatically generated by a compiler. The CBL works by extending SGML/XML, which is normally used to formally define the structures of particular document types, to include specification of the semantics associated with each information element and attribute. The limited set of (mostly) character types in SGML/XML can be extended to declare any kind of datatype.

Here is a fragment from the CBL definition for the "datetime" module:

```
<!-- datetime.mod Version: 1.0 -->
<!-- Copyright 1998 Veo Systems, Inc. -->
...
<!ELEMENT year (#PCDATA)>
<!ATTLIST year
        schema CDATA #FIXED "urn:x-veosystems:stds:iso:8601:3.8"
>
<!ELEMENT month (#PCDATA)>
<!ATTLIST month
        schema CDATA #FIXED "url:x-veosystems:stds:iso:8601:3.12"
>
...
```

In this fragment, the ELEMENT "year" is defined as character data, and an associated "schema" attribute, also character data, defines the schema for "year" to be section 3.8 of the ISO 8601 standard.

This "datetime" CBL module is in fact defined as an instance of the Schema DTD. First, the module name is defined. Then the "datetime" element "YEAR" is bound to the semantics of ISO 8601:

```
<!DOCTYPE SCHEMA SYSTEM "schema.dtd">
<SCHEMA><H1>Date and Time Module</H1>
...
<ELEMNTTYPE NAME="year"DATATYPE="YEAR"><MODEL>
        <STRING
DATATYPE="YEAR"></STRING></MODEL>
<ATTDEF NAME=:schema:iso8601" DATATYPE="CDATA">
        <FIXED>3.8
Gregorian calendar</FIXED></ATTDEF></ELEMENTTYPE>
...
```

The example market participant and service modules above are also stored in the CBL repository.

In FIG. **10**, an Airbill **1000** is being defined by customizing a generic purchase order DTD **1001**, adding more specific information about shipping weight **1002**. The generic purchase order **1001** was initially assembled from the ground up out of CBL modules for address, date and time, currency, and vendor and product description. Using CBL thus significantly speeds the development and implementation of XML commerce applications. More importantly, CBL makes it easier for commercial applications to be interconnected.

In the CBL, XML is extended with a schema. The extensions add strong-typing to XML elements so that content can be readily validated. For example, an element called <CPU_clock_speed> can be defined as an integer with a set of valid values: {100, 133, 166, 200, 233, 266 Mhz.}. The schema also adds class-subclass hierarchies, so that information can be readily instantiated from class definitions. A laptop, for instance, can be described as a computer with additional tags for features such as display type and battery life. These and other extensions facilitate data entry, as well as automated translations between XML and traditional Object-Oriented and relational data models.

Thus the completed BID is run through the compiler which produces the DTDs for the actual instance of a participant and a service as outlined above, the JAVA beans which correspond to the logical structures in the DTD instances, and transformation code for transforming from XML to JAVA and from JAVA to XML. In alternative systems documentation is also generated for display on a user interface or for printing by a user to facilitate use of the objects.

For the example market participant and service DTDs set forth above, the JAVA beans generated by the compiler are set forth (with some redactions for conciseness) as follows:

```
import com.veo.vsp.doclet.meta.Document;
public class AddressPhysical extends Document {
    public static final String DOC_TYPE = "address.physical";
     String mStreet;
     String mCity;
    public final static int AK = 0;
    public final static int AL = 1;
    public final static int AR = 2;
    public final static int AZ = 3;
    public final static int CA = 4;
    public final static int WI = 48;
    public final static int WV = 49;
    public final static int WY = 50;
     int mState;
     String mPostcode;
    public final static int AD = 51;
    public final static int AE = 52;
    public final static int AF = 53;
    public final static int AG = 54;
    public final static int AI = 55;
    public final static int AM = 56;
    . . .
     int mCountry;
    public AddressPhysical( ){
            super(DOC_TYPE);
    mStreet = new String( );
    mCity = new String( );
            this.mState = -1;
    mPostcode = null;
            this.mCountry= -1;
    }
    public AddressPhysical(String doc_type){
            super(doc_type);
    mStreet = new String( );
    mCity = new String( );
            this.mState = -1;
    mPostcode = null;
            this.mCountry = -1;
    }
    static public AddressPhysical initAddressPhysical(String iStreet,String iCity,int
iState,String iPostcode,int iCountry){
            AddressPhysical obj = new AddressPhysical( );
            obj.initializeAll(iStreet, iCity, iState, iPostcode, iCountry);
            return obj;
    }
    public void initializeAll(String iStreet,String iCity,int iState,String iPostcode,int
iCountry){
            mStreet = iStreet;
            mCity = iCity;
            mState = iState;
            mPostcode = iPostcode;
            mCountry = iCountry;
    }
    public String getStreet( ){
     return mStreet;
    }
    public String getStreetToXML( ){
            if (getStreet( ) == null) return null;
            char [ ] c = getStreet( ).toCharArray( );
            StringBuffer sb = new StringBuffer( );
            for (intx = 0; x < c.length; x++){
             switch(c[x]){
             case '>':
              sb.append("&gt;");
```

-continued

```
                break;
              case '<':
                sb.append("&lt;");
                break;
              case '&':
                sb.append("&amp;");
                break;
              case '"':
                sb.append("&quot;");
                break;
              case '\'':
                sb.append("&quot;");
                break;
              default:
                if (Character.isDefined(c[x]))
                  sb.append(c[x]);
              }
            }
            return sb.toString( );
}
public void setStreet(String inp){
            this.mStreet = inp;
}
public void streetFromXML(String n){
            setStreet(n);
}
public String getCity( ){
  return mCity;
}
public String getCityToXML( ){
            if (getCity( ) == null) return null;
            char [ ] c = getCity( ).toCharArray( );
            StringBuffer sb = new StringBuffer( );
            for (int x = 0; x < c.length; x++){
              switch(c[x]){
              case '>':
                sb.append("&gt;");
                break;
              case '<':
                sb.append("&lt;");
                break;
              case '&':
                sb.append("&amp;");
                break;
              case '"':
                sb.append("&quot;");
                break;
              case '\'':
                sb.append("&quot;");
                break;
              default:
                if (Character.isDefined(c[x]))
                  sb.append(c[x]);
              }
            }
            return sb.toString( );
}
public void setCity(String inp){
            this.mCity = inp;
}
public void cityFromXML(String n){
            setCity(n);
}
public int getState( ){
  return mState;
}
public String getStateToXML( ){
            switch (mState) {
                        case AK: return "AK";
                        case AL: return "AL";
                        case AR: return "AR";
                        case AZ: return "AZ";
                        . . .
            }
  return null;
}
public void setState(int inp){
            this.mState = inp;
}
```

-continued

```
public void stateFromXML(String s){
        if (s.equals("AK")) mState = AK;
        else if (s.equals("AL"))mState = AL;
        else if (s.equals("AR"))mState = AR;
        else if (s.equals("AZ"))mState = AZ;
        . . .
}
public String getPostcode( ){
 return mPostcode;
}
public String getPostcodeToXML( ){
        if (getPostcode( ) == null) return null;
        char [ ] c = getPostcode( ).toCharArray( );
        StringBuffer sb = new StringBuffer( );
        for (int x = 0; x < c.length; x++){
         switch(c[x]){
         case '>':
          sb.append("&gt;");
          break;
         case '<':
          sb.append("&lt;");
          break;
         case '&':
          sb.append("&amp;");
          break;
         case '"':
          sb. append("&quot;");
          break;
         case '\'':
          sb.append("&quot;");
          break;
         default:
          if (Character.isDefined(c[x]))
           sb.append(c[x]);
         }
        }
        return sb.toString( );
}
public void setPostcode(String inp){
        this.mPostcode = inp;
}
public void postcodeFromXML(String n){
        setPostcode(n);
}
public int getCountry( ){
 return mCountry;
}
public String getCountryToXML( ){
        switch (mCountry){
                case AD: return "AD";
                case AE: return "AE";
                case AF: return "AF"
                . . .
        }
 return null;
}
public void setCountry(int inp){
        this.mCountry = inp;
}
public void countryFromXML(String s){
        if(s.equals("AD")) mCountry = AD;
        else if (s.equals("AE"))mCountry = AE;
        else if (s.equals("AF"))mCountry = AF;
        else if (s.equals("AG"))mCountry = AG;
        else if (s.equals("AI"))mCountry = AI;
        . . .
        }
}
package com.veo.xdk.dev.schema.test.blib;
import com.veo.vsp.doclet.meta.Document;
public class AddressSet extends Document {
        public static final String DOC_TYPE = "address.set";
        AddressPhysical mAddressPhysical;
         String [ ] mTelephone;
         String [ ] mFax;
         String [ ] m Email;
         String [ ] mInternet;
        public AddressSet( ){
                super(DOC_TYPE);
```

-continued

```
                this.mAddressPhysical = new AddressPhysical( );
        mTelephone = null;
        mFax = null;
        mEmail = null;
        mInternet = null;
        }
        public AddressSet(String doc_type){
                super(doc_type);
                this.mAddressPhysical = new AddressPhysical( );
        mTelephone = null;
        mFax = null;
        mEmail = null;
        mInternet = null;
        }
        static public AddressSet initAddressSet(AddressPhysical iAddressPhysical,String [ ]
iTelephone,String [ ] iFax,String [ ] iEmail,String [ ] iInternet){
                AddressSet obj = new AddressSet( );
                obj.initializeAll(AddressPhysical, iTelephone, iFax, iEmail, iInternet);
                return obj;
        }
        public void initializeAll(AddressPhysical iAddressPhysical,String [ ] iTelephone,String [ ]
iFax,String [ ] iEmail,String [ ] iInternet){
                mAddressPhysical = iAddressPhysical;
                mTelephone = iTelephone;
                mFax = iFax;
                mEmail = iEmail;
                mInternet = iInternet;
        }
        public AddressPhysical getAddressPhysical( ){
         return mAddressPhysical;
        }
        public void setAddressPhysical(AddressPhysical inp){
                this.mAddressPhysical = inp;
        }
        public String [ ] getTelephone( ){
         return mTelephone;
        }
        public String getTelephone(int index){
                if (this.mTelephone = null)
                 return null;
                if (index >= this.mTelephone.length)
                 return null;
                if (index <0 && -index> this.mTelephone.length)
                 return null;
                if (index >= 0) return this.mTelephone[index];
                return this.mTelephone[this.mTelephone.length + index];
        }
        public String [ ] getTelephoneToXML( ){
                String [ ] valArr = getTelephone( );
                if (valArr == null) return null;
                String [ ] nvArr = new String[valArr.length];
                for (int z = 0; z < nvArr.length; z++){
                char [ ] c = valArr[z].toCharArray( );
                StringBuffer st = new StringBuffer( );
                StringBuffer sb = new StringBuffer( );
                for (int x = 0; x < c.length; x++){
                 switch(c[x]){
                 case '>':
                  sb.append("&gt;");
                  break;
                 case '<':
                  sb.append("&lt;");
                  break;
                 case '&':
                  sb.append("&amp;");
                  break;
                 case '"':
                  sb.append("&quot;");
                  break;
                 case '"':
                  sb. append("&quot;");
                  break;
                 default:
                  if (Character.isDefined(c[x]))
                   sb. append(c[x]);
                 }
                }
                nvArr[z] = sb.toString( );
                }
```

-continued

```
            return nvArr;
        }
        public void setTelephone(int index, String inp){
            if (this.mTelephone == null){
                if(index<0) {
                 this.mTelephone = new String[1];
                 this.mTelephone[0] = inp;
                } else {
                 this.mTelephone = new String[index + 1];
                 this.mTelephone[index] = inp;
                }
            } else if (index < 0) {
                    String [ ] newTelephone = new String[this.mTelephone.length + 1];
                    java.lang.System.arraycopy((Object)mTelephone, 0,
(Object)newTelephone, 0, this.mTelephone.length);
                    newTelephone[newTelephone.length - 1] = inp;
                    mTelephone = newTelephone;
                } else if (index >= this.mTelephone.length){
                    String [ ] newTelephone = new String[index + 1];
                    java.lang.System.arraycopy((Object)mTelephone, 0,
(Object)newTelephone, 0, this.mTelephone.length);
                    newTelephone[index] = inp;
                    mTelephone = newTelephone;
                } else {
                    this.mTelephone[index] = inp;
                }
        }
        public void setTelephone(String [ ] inp){
            this.mTelephone = inp;
        }
        public void telephoneFromXML(String n){
            setTelephone(-1, n);
        }
        public String [ ] getFax( ){
         return mFax;
        }
        public String getFax(int index){
            if(this.mFax == null)
             return null;
            if(index >= this.mFax.length)
             return null;
            if (index < 0 && -index> this.mFax.length)
             return null;
            if (index >= 0) return this.mFax[index];
            return this.mFax[this.mFax.length + index];
        }
        public String [ ] getFaxToXML( ){
            String [ ] valArr = getFax( );
            if (valArr == null) return null;
            String [ ] nvArr = new String[valArr.length];
            for (int z = 0; z < nvArr.length; z++){
            char [ ] c = valArr[z].toCharArray( );
            StringBuffer st = new StringBuffer( );
            StringBuffer sb = new StringBuffer( );
            for (int x = 0; x < c.length; x++){
             switch(c[x]){
             case '>':
               sb.append("&gt;");
               break;
             case '<':
               sb.append("&lt;");
               break;
             case '&':
               sb.append("&amp;");
               break;
             case '"':
               sb.append("&quot;");
               break;
             case '':
               sb.append("&quot;");
               break;
             default:
               if (Character.isDefined(c[x]))
                sb.append(c[x]);
             }
            }
            nvArr[z] = sb.toString( );
            }
            return nvArr;
```

-continued

```
        }
        public void setFax(int index, String inp){
          if(this.mFax == null){
                if(index < 0) {
                 this.mFax = new String[1];
                 this.mFax[0] = inp;
                } else {
                 this.mFax = new String[index + 1];
                 this.mFax[index] = inp;
                }
          } else if (index < 0) {
                    String [ ] newFax = new String[this.mFax.length + 1];
                    java.lang.System.arraycopy((Object)mFax, 0, (Object)newFax, 0,
this.mFax.length);
                    newFax[newFax.length - 1] = inp;
                    mFax = newFax;
                } else if (index >= this.mFax.length){
                    String [ ] newFax = new String[index + 1];
                    java.lang.System.arraycopy((Object)mFax, 0, (Object)newFax, 0,
this mFax length);
                    newFax[index] = inp;
                    mFax = newFax
                } else{
                    this.mFax[index] = inp;
                }
        }
        public void setFax(String [ ] inp){
                this.mFax = inp;
        }
        public void faxFromXML(String n){
                setFax(-1, n);
        }
        public String [ ] getEmail( ){
         return mEmail;
        }
        public String getEmail(int index){
                if (this.mEmail = null)
                 return null;
                if (index >= this.mEmail.length)
                 return null;
                if (index < 0 && -index> this.mEmail.length)
                 return null;
                if (index >= 0) return this.mEmail[index];
                return this.Email[this.Email.length + index];
        }
        public String [ ] getEmailToXML( ){
                String [ ] valArr = getEmail( );
                if (valArr == null) return null;
                String [ ] nvArr = new String[valArr.length];
                for (int z = 0; z < nvArr.length; z++){
                char [ ] c = valArr[z].toCharArray( );
                StringBuffer st = new StringBuffer( );
                StringBuffer sb = new StringBuffer( );
                for (int x = 0; x < c.length; x++){
                 switch(c[x]){
                 case '>':
                  sb. append("&gt;");
                  break;
                 case '<':
                  sb.append("&lt;");
                  break;
                 case '&':
                  sb.append("&amp;");
                  break;
                 case '"':
                  sb. append("&quot;");
                  break;
                 case '"':
                  sb.append("&quot;");
                  break;
                 default:
                  if (Character.isDefined(c[x]))
                   sb.append(c[x]);
                 }
                }
                nvArr[z] = sb.toString( );
                }
                return nvArr;
        }
```

-continued

```
public void setEmail(int index, String inp){
if(this.mEmail == null) {
    if(index < 0) {
     this.mEmail = new String[1];
     this.mEmail[0] = inp;
    } else {
     this.mEmail = new String[index + 1];
     this.mEmail[index] = inp;
    }
} else if (index < 0) {
        String [ ] newEmail = new String[this.mEmail.length + 1];
        java.lang.System.arraycopy((Object)mEmail, 0, (Object)newEmail, 0,
this.mEmail.length);
        newEmail[newEmail.length - 1] = inp;
        mEmail = newEmail;
    } else if (index >= this.mEmail.length){
        String [ ] newEmail = new String[index + 1];
        java.lang.System.arraycopy((Object)mEmail, 0, (Object)newEmail, 0,
this.mEmail.length);
        newEmail[index] = inp;
        mEmail = newEmail;
    } else {
        this.mEmail[index] = inp;
    }
}
public void setEmail(String [ ] inp){
    this.mEmail = inp;
}
public void emailFromXML(String n){
    setEmail(-1, n);
}
public String [ ] getInternet( ){
 return mInternet;
}
public String getInternet(int index){
    if (this.mInternet == null)
     return null;
    if (index >= this.mInternet.length)
     return null;
    if (index < 0 && -index > this.mInternet.length)
     return null;
    if (index >= 0) return this.mInternet[index];
    return this.mInternet[this.mInternet.length + index];
}
public String [ ] getInternetToXML( ){
    String [ ] valArr = getInternet( );
    if (valArr == null) return null;
    String [ ] nvArr = new String[valArr.length];
    for (int z = 0; z < nvArr.length; z++){
    char [ ] c = valArr[z].toCharArray( );
    StringBuffer st = new StringBuffer( );
    StringBuffer sb = new StringBuffer( );
    for (int x = 0; x < c.length; x++){
     switch(c[x]){
     case '>':
      sb.append("&gt;");
      break;
     case '<':
      sb.append("&lt;");
      break;
     case '&':
      sb.append("&amp;");
      break;
     case '"':
      sb.append("&quot;");
      break;
     case '"':
      sb.append("&quot;");
      break;
     default:
      if (Character.isDefined(c[x]))
       sb.append(c[x]);
     }
    }
    nvArr[z] sb.toString( );
    }
    return nvArr;
}
public void setInternet(int index, String inp){
```

-continued

```
            if (this.mInternet == null){
              if(index < 0) {
               this.mInternet = new String[1];
               this.mInternet[0] = inp;
              } else {
               this.mInternet new String[index + 1];
               this.mInternet[index] = inp;
              }
            } else if (index < 0) {
                  String [ ] newInternet = new String[this.mInternet.length + 1];
                  java.lang.System.arraycopy((Object)mInternet, 0, (Object)newInternet,
0, this.mInternet.length);
                  newInternet[newInternet.length - 1] = inp;
                  mInternet = newInternet;
              } else if (index >= this.mInternet,length){
                  String [ ] newInternet = new String[index + 1];
                  java.lang.System.arraycopy((Object)mInternet, 0, (Object)newInternet,
0, this.mInternet.length);
                  newInternet[index] = inp;
                  mInternet = newInternet;
              } else {
                  this.mInternet[index] = inp;
              }
        }
        public void setInternet(String [ ] inp){
                  this.mInternet = inp;
        }
        public void internetFromXML(String n){
                  setInternet(-1, n);
        }
}
package com.veo.xdk.dev.schema.test.blib;
import com.veo.vsp.doclet.meta.Document;
public class Business extends Party {
        public static final String DOC_TYPE = "business";
         String aBusinessNumber;
        public Business( ){
                  super(DOC_TYPE);
        aBusinessNumber = new String( );
        }
        public Business(String doc_type){
                  super(doc_type);
        aBusinessNumber = new String( );
        }
        static public Business initBusiness(String iBusinessNumber,String [ ]
iPartyName,AddressSet iAddressSet){
                  Business obj = new Business( );
                  obj.initializeAll(iBusinessNumber, iPartyName, iAddressSet);
                  return obj;
        }
        public void initializeAll(String iBusinessNumber,String [ ] iPartyNameAddressSet
iAddressSet){
                  aBusinessNumber = iBusinessNumber;
                  super.initializeAll(iPartyName, iAddressSet);
        }
        public String getBusinessNumber( ){
         return aBusinessNumber;
        }
        public String getBusinessNumberToXML( ){
                  if (getBusinessNumber( ) == null) return null;
                  char [ ] c = getBusinessNumber( ).toCharArray( );
                  StringBuffer sb = new StringBuffer( );
                  for (int x = 0; x < c.length; x++){
                   switch(c[x]){
                   case '>':
                    sb.append("&gt;");
                    break;
                   case '<':
                    sb. append("&lt;");
                    break;
                   case '&':
                    sb.append("&amp;");
                    break;
                   case '"':
                    sb.append("&quot;");
                    break;
                   case '\':
                    sb.append("&quot;");
                    break;
```

-continued

```
                    default:
                     if (Character.isDefined(c[x]))
                      sb.append(c[x]);
                    }
                   }
                   return sb.toString( );
          }
          public void setBusinessNumber(String inp){
                   this.aBusinessNumber = inp;
          }
          public void businessNumberFromXML(String n){
                   setBusinessNumber(n);
          }
}
import com.veo.vsp.doclet.meta.Document;
public class Party extends Document{
          public static final String DOC_TYPE = "party";
          String [ ] mPartyName;
          AddressSet mAddressSet;
          public Party( ){
                   super(DOC_TYPE);
          mPartyName = new String[0];
                   this.mAddressSet = new AddressSet( );
          }
          public Party(String doc_type){
                   super(doc_type);
          mPartyName = new String[0];
                   this.mAddressSet = new AddressSet( );
          }
          static public Party initParty(String [ ] iPartyName,AddressSet iAddressSet){
                   Party obj = new Party( );
                   obj.initializeAll(iPartyName, iAddressSet);
                   return obj;
          }
          public void initializeAll(String [ ] iPartyName,AddressSet iAddressSet){
                   mPartyName = iPartyName;
                   mAddressSet = iAddressSet;
          }
          public String [ ] getPartyName( ){
           return mPartyName;
          }
          public String getPartyName(int index){
                   if (this.mPartyName == null)
                    return null;
                   if (index >= this.mPartyName.length)
                    return null
                   if (index < 0 && -index> this.mPartyName.length)
                    return null;
                   if(index >= 0) return this.mPartyName[index];
                   return this.mPartyName[this.mPartyName.length + index];
          }
          public String [ ] getPartyNameToXML( ){
                   String [ ] valArr = getPartyName( );
                   if (valArr == null) return null;
                   String [ ] nvArr = new String[valArr.length];
                   for (int z = 0; z < nvArr.length; z++){
                   char [ ] c = nvArr[z].toCharArray( );
                   StringBuffer st = new StringBuffer( );
                   StringBuffer sb = new StringBuffer( );
                   for (int x = 0; x < c.length; x++){
                    switch(c[x]){
                    case '>'
                     sb.append("&gt;");
                     break;
                    case '<':
                     sb.append("&lt;");
                     break;
                    case '&':
                     sb.append("&amp;");
                     break;
                    case '"':
                     sb.append("&quot;");
                     break;
                    case '"':
                     sb.append("&quot;");
                     break;
                    default:
                     if (Character.isDefined(c[x]))
                      sb.append(c[x]);
```

-continued

```
                    }
                   }
                   nvArr[z] = sb.toString( );
                   }
                   return nvArr;
          }
          public void setPartyName(int index, String inp){
            if (this.mPartyName == null) {
              if(index < 0) {
               this.mPartyName = new String[1];
               this.mPartyName[0] = inp;
              } else {
               this.mPartyName = new String[index + 1];
               this.mPartyName[index] = inp;
              }
            } else if (index < 0) {
                   String [ ] newPartyName = new String[this.mPartyName.length + 1];
                   java.lang.System.arraycopy((Object)mPartyName, 0,
(Object)newPartyName, 0, this.mPartyName.length);
  newPartyName[newPartyName.length - 1] = inp;
  mPartyName = newPartyName;
               } else if (index >= this.mPartyName.length){
                   String [ ] newPartyName = new String[index + 1];
                   java.lang.System.arraycopy((Object)mPartyName, 0,
(Object)newPartyName, 0, this.mPartyName.length);
                   newPartyName[index] = inp;
                   mPartyName = newPartyName;
               } else {
                   this.mPartyName[index] = inp;
               }
          }
          public void setPartyName(String [ ] inp){
               this.mPartyName = inp;
          }
          public void PartyNameFromXML(String n){
               setPartyName(-1, n);
          }
          public AddressSet getAddressSet( ){
           return mAddressSet;
          }
          public void setAddressSet(AddressSet inp){
               this.mAddressSet = inp;
          }
}
package com.veo.xdk.dev.schema.test.blib;
import com.veo.vsp.doclet,meta.Document;
public class Person extends Party {
          public static final String DOC_TYPE = "person";
           String aSSN;
          public Person( ){
               super(DOC_TYPE);
          aSSN = null;
          }
          public Person(String doc_type){
               super(doc_type);
          aSSN = null;
          }
          static public Person initPerson(String iSSN,String [ ] iPartyName,AddressSet
iAddressSet){
               Person obj = new Person( );
               obj.initializeAll(iSSN, iPartyName, iAddressSet);
               return obj;
          }
          public void initializeAll(String iSSN,String [ ] iPartyName,AddressSet iAddressSet){
               aSSN = iSSN;
               super.initializeAll(iPartyName, iAddressSet);
          }
          public String getSSN( ){
           return aSSN;
          }
          public String getSSNToXML( ){
               if (getSSN( ) == null) return null;
               char [ ] c = getSSN( ).toCharArray( );
               StringBuffer sb = new StringBuffer( );
               for (int x = 0; x < c.length; x++){
                switch(c[x]){
                case '>':
                 sb.append("&gt;");
                 break;
```

-continued

```
                case '<':
                 sb.append("&lt;");
                 break;
                case '&':
                 sb.append("&amp;");
                 break;
                case '"':
                 sb.append("&quot;");
                 break;
                case '\'':
                 sb.append("&quot;");
                 break;
                default:
                 if (Character.isDefined(c[x]))
                  sb.append(c[x]);
                }
               }
               return sb.toString( );
          }
          public void setSSN(String inp){
               this.aSSN = inp;
          }
          public void sSNFromXML(String n){
               setSSN(n);
          }
}
package com.veo.xdk.dev.schema.test.blib;
import com.veo.vsp.doclet.meta.Document;
public class PrototypeService extends Document{
          public static final String DOC_TYPE = "prototype.service";
           String mServiceName;
           String [ ] mServiceTerms;
           String [ ] mServiceLocation;
           ServiceOperation [ ] mServiceOperation;
          public PrototypeService( ){
               super(DOC_TYPE);
          mServiceName = new String( );
          mServiceTerms = new String[0];
          mServiceLocation = new String[0];
               this.mServiceOperation = new ServiceOperation[0];
          }
          public PrototypeService(String doc_type){
               super(doc_type);
          mServiceName = new String( );
          mServiceTerms = new String[0];
          mServiceLocation = new String[0];
               this.mServiceOperation = new ServiceOperation[0];
          }
          static public PrototypeService initPrototypeService(String iServiceName,String [ ]
iServiceTerms,String [ ] iServiceLocation,ServiceOperation [ ] iServiceOperation){
               PrototypeService obj = new PrototypeService( );
               obj.initializeAll(iServiceName, iServiceTerms, iServiceLocation,
iServiceOperation);
               return obj;
          }
          public void initializeAll(String iServiceName,String [ ] iServiceTerms,String [ ]
iServiceLocation,ServiceOperation [ ] iServiceOperation){
               mServiceName = iServiceName;
               mServiceTerms = iServiceTerms;
               mServiceLocation = iServiceLocation;
               mServiceOperation = iServiceOperation;
          }
          public String getServiceName( ){
           return mServiceName;
          }
          public String getServiceNameToXML( ){
               if (getServiceName( ) == null) return null;
               char [ ] c = getServiceName( ).toCharArray( );
               StringBuffer sb = new StringBuffer( );
               for (int x = 0; x < c.length; x++){
                switch(c[x]){
                case '>':
                 sb.append("&gt;");
                 break;
                case '<':
                 sb.append("&lt;");
                 break;
                case '&':
                 sb.append("&amp;");
```

-continued

```
            break;
           case '"':
            sb.append("&quot;");
            break;
           case '\':
            sb.append("&quot;");
            break;
           default:
            if (Character.isDefined(c[x]))
             sb.append(c[x]);
           }
          }
          return sb.toString( );
    }
    public void setServiceName(String inp){
          this.mServiceName = inp;
    }
    public void serviceNameFromXML(String n){
          setServiceName(n);
    }
    public String [ ] getServiceTerms( ){
     return mServiceTerms;
    }
    public String getServiceTerms(int index){
          if (this.mServiceTerms == null)
           return null;
          if (index >= this.mServiceTerms.length)
           return null;
          if (index < 0 && -index>this.mServiceTerms.length)
           return null;
          if (index >= 0) return this.mServiceTerms[index];
          return this.mServiceTerms[this.mServiceTerms.length + index];
    }
    public String [ ] getServiceTermsToXML( ){
          String [ ] valArr = getServiceTerms( );
          if (valArr == null) return null;
          String [ ] nvArr = new String[valArr.length];
          for (int z = 0; z < nvArr.length; z++){
          char [ ] c = valArr[z].toCharArray( );
          StringBuffer st = new StringBuffer( );
          StringBuffer sb = new StringBuffer( );
          for (int x = 0; x < c.length; x++){
           switch(c[x]){
           case '>':
            sb.append("&gt;");
            break;
           case '<':
            sb.append("&lt;");
            break;
           case '&':
            sb.append("&amp;");
            break;
           case '"':
            sb.append("&quot;");
            break;
           case '"':
            sb.append("&quot;");
            break;
           default:
            if (Character.isDefined(c[x]))
             sb.append(c[x]);
           }
          }
          nvArr[z] = sb.toString( );
          }
          return nvArr;
    }
    public void setServiceTerms(int index, String inp){
       if (this.mServiceTerms == null){
          if(index < 0) {
           this.mServiceTerms = new String[1];
           this.mServiceTerms[0] = inp;
          } else {
           this.inServiceTerms = new String[index + 1];
           this.mServiceTerms[index] = inp;
          }
       } else if (index < 0) {
              String [ ] newServiceTerms = new String[this.mServiceTerms.length +
1];
```

-continued

```
                    java.lang.System.arraycopy((Object)mServiceTerms; 0,
(Object)newServiceTerms, 0, this.mServiceTerms.length);
                    newServiceTerms[newServiceTerms.length - 1] = inp;
                    mServiceTerms = newServiceTerms;
                } else if (index >= this.mServiceTerms.length){
                    String [ ] newServiceTerms = new String[index + 1];
                    java.lang.System.arraycopy((Object)mServiceTerms, 0,
(Object)newServiceTerms, 0, this.mServiceTerms.length);
                    newServiceTerms[index] = inp;
                    mServiceTerms = newServiceTerms,
                } else {
                    this.mServiceTerms[index] = inp,
                }
            }
            public void setServiceTerms(String [ ] inp){
                this.mServiceTerms = inp;
            }
            public void serviceTermsFromXML(String n){
                setServiceTerms(-1, n);
            }
            public String [ ] getServiceLocation( ){
              return mServiceLocation;
            }
            public String getServiceLocation(int index){
                if (this.mServiceLocation == null)
                  return null;
                if (index >= this.mServiceLocation.length)
                  return null;
                if (index < 0 && -index>this.mServiceLocation.length)
                  return null;
                if (index >= 0) return this.mServiceLocation[index];
                return this.mServiceLocation[this.mServiceLocation.length + index);
            }
            public String [ ] getServiceLocationToXML( ){
                String [ ] valArr = getServiceLocation( );
                if (valArr == null) return null;
                String [ ] nvArr = new String[valAr.length];
                for (int z = 0; z < nvArr.length; z++){
                char [ ] c = valArr[z].toCharArray( );
                StringBuffer st = new StringBuffer( );
                StringBuffer sb = new StringBuffer( );
                for (int x = 0; x < c.length; x++){
                  switch(c[x]){
                  case '>':
                    sb.append("&gt;");
                    break;
                  case '<':
                    sb.append("&lt;");
                    break;
                  case '&':
                    sb.append("&amp;");
                    break;
                  case '"':
                    sb.append("&quot;");
                    break;
                  case '"':
                    sb.append("&quot;");
                    break;
                  default:
                    if (Character.isDefined(c[x]))
                      sb.append(c[x]);
                  }
                }
                nvArr[z] = sb.toString( );
                }
                return nvArr;
            }
            public void setServiceLocation(int index, String inp){
              if (this.mServiceLocation == null) {
                if(index < 0) {
                  this.mServiceLocation = new String[1];
                  this.mServiceLocation[0] = inp;
                } else{
                  this.mServiceLocation = new String[index + 1];
                  this.mServiceLocation[index] = inp;
                }
            } else if (index < 0) {
                    String [ ] newServiceLocation = new
String[this.mServiceLocation.length + 1];
```

-continued

```
                    java.lang.System.arraycopy((Object)mServiceLocation, 0,
(Object)newServiceLocation, 0, this.mServiceLocation.length);
                    newServiceLocation[newServiceLocation.length - 1] = inp;
                    mServiceLocation = newServiceLocation;
               } else if (index > this.mServiceLocation.length){
                    String [ ] newServiceLocation = new String[index + 1];
                    java.lang.System.arraycopy((Object)mServiceLocation, 0,
(Object)newServiceLocation, 0, this.mServiceLocation.length);
                    newServiceLocation[index] = inp;
                    mServiceLocation = newServiceLocation;
               } else {
                    this.mServiceLocation[index] = inp;
               }
          }
          public void setServiceLocation(String [ ] inp){
               this.mServiceLocation = inp;
          }
          public void serviceLocationFromXML(String n){
               setServiceLocation(-1, n);
          }
          public ServiceOperation [ ] getServiceOperation( ){
            return mServiceOperation;
          }
          public ServiceOperation getServiceOperation(int index){
               if (this.mServiceOperation == null)
                 return null;
               if (index >= this.mServiceOperation.length)
                 return null;
               if (index < 0 && -index >= this.mServiceOperation.length)
                 return null;
               if (index >= 0) return this.mServiceOperation[index];
               return this.mServiceOperation[this.mServiceOperation.length + index];
          }
          public void setServiceOperation(int index, ServiceOperation inp){
            if (this.mServiceOperation = null){
               if(index < 0) {
                 this.mServiceOperation = new ServiceOperation[1];
                 this.mServiceOperation[0] = inp;
               } else {
                 this.mServiceOperation = new ServiceOperation[index + 1];
                 this.mServiceOperation[index] = inp;
               }
            } else if (index < 0) {
                    ServiceOperation [ ] newServiceOperation = new
ServiceOperation[this.mServiceOperation.length + 1];
                    java.lang.System.arraycopy((Object)mServiceOperation, 0,
(Object)newServiceOperation, 0, this.mServiceOperation.length);
                    newServiceOperation[newserviceOperation.length - 1] = inp;
                    mServiceOperation = newServiceOperation;
               } else if (index >= this.mServiceOperation.length){
                    ServiceOperation [ ] newServiceOperation = new
ServiceOperation[index + 1];
                    java.lang.System.arraycopy((Object)mServiceOperation, 0,
(Object)newServiceOperation, 0, this.mServiceOperation.length);
                    newServiceOperation[index] = inp;
                    mServiceOperation = newServiceOperation;
               } else {
                    this.mServiceOperation[index] = inp;
               }
          }
          public void setServiceOperation(ServiceOperation [ ] inp){
               this.mServiceOperation = inp;
          }
}
package com.veo.xdk.dev.schema.test.blib;
import com.veo.vsp.doclet.meta.Document;
public class Service extends Document{
          public static final String DOC_TYPE = "service";
          String mServiceName;
          String mServiceLocation;
          ServiceOperation [ ] mServiceOperation;
          String mServiceTerms;
          public Service( ){
               super(DOC_TYPE);
          mServiceName = new String( );
          mServiceLocation = new String( );
               this.mServiceOperation = new ServiceOperation[0];
          mServiceTerms = new String( );
          }
```

-continued

```
        public Service(String doc_type){
            super(doc_type);
        mServiceName = new String( );
        mServiceLocation = new String( );
            this.mServiceOperation = new ServiceOperation[0];
        mServiceTerms = new String( );
        }
        static public Service initService(String iServiceName,String
iServiceLocation,ServiceOperation [ ] iServiceOperation,String iServiceTerms){
            Service obj = new Service( );
            obj.initializeAll(iServiceName, iServiceLocation, iServiceOperation,
iServiceTerms);
            return obj;
        }
        public void initializeAll(String iServiceName,String iServiceLocation,ServiceOperation
[ ] iServiceOperation,String iServiceTerms){
            mServiceName = iServiceName;
            mServiceLocation = iServiceLocation;
            mServiceOperation = iServiceOperation;
            mServiceTerms = iServiceTerms;
        }
        public String getServiceName( ){
         return mServiceName;
        }
        public String getServiceNameToXML( ){
            if (getServiceName( ) == null) return null;
            char [ ] c = getServiceName( ).toCharArray( );
            StringBuffer sb = new StringBuffer( );
            for (int x= 0; x < c.length; x++){
             switch(c[x]){
             case '>':
              sb. append("&gt;");
              break;
             case '<':
              sb.append("&lt;");
              break;
             case '&':
              sb.append("&amp;");
              break;
             case '"':
              sb. append("&quot;");
              break;
             case '\'':
              sb.append("&quot;");
              break;
             default:
              if (Character.isDefined(c[x]))
               sb.append(c[x]);
             }
            }
            return sb.toString( );
        }
        public void setServiceName(String inp){
            this.mServiceName = inp;
        }
        public void serviceNameFromXML(String n){
            setServiceName(n);
        }
        public String getServiceLocation( ){
         return mServiceLocation;
        }
        public String getServiceLocationToXML( ){
            if (getServiceLocation( ) == null) return null;
            char [ ] c = getServiceLocation( ).toCharArray( );
            StringBuffer sb = new StringBuffer( );
            for (int x = 0; x < c.length; x++){
             switch(c[x]){
             case '>':
              sb.append("&gt;");
              break;
             case '<':
              sb.append("&lt;");
              break;
             case '&':
              sb.append("&amp;");
              break;
             case '"':
              sb.append("&quot;");
              break;
```

-continued

```
                case '\':
                 sb.append("&quot;");
                 break;
                default:
                 if (Character.isDefined(c[x]))
                  sb.append(c[x]);
                }
               }
               return sb.toString( );
          }
          public void setServiceLocation(String inp){
               this.mServiceLocation = inp;
          }
          public void serviceLocationFromXML(String n){
               setServiceLocation(n);
          }
          public ServiceOperation [ ] getServiceOperation( ){
           return mServiceOperation;
          }
          public ServiceOperation getServiceOperation(int index){
               if (this.mServiceOperation == null)
                return null;
               if (index >= this.mServiceOperation.length)
                return null;
               if (index < 0 && -index > this.mServiceOperation.length)
                return null;
               if (index >= 0) return this.mServiceOperation[index];
               return this.mServiceOperation[this.mServiceOperation.length + index];
          }
          public void setServiceOperation(int index, ServiceOperation inp){
            if (this.mServiceOperation == null) {
               if (index < 0) {
                this.mServiceOperation = new ServiceOperation[1];
                this.mServiceOperation[0] = inp;
               } else {
                this.mServiceOperation = new ServiceOperation[index + 1];
                this.mServiceOperation[index] = inp;
               }
            } else if (index < 0) {
                    ServiceOperation [ ] newServiceOperation = new
ServiceOperation[this.mServiceOperation.length + 1];
                    java.lang.System.arraycopy((Object)mServiceOperation, 0,
(Object)newServiceOperation, 0, this.mServiceOperation.length);
                    newServiceOperation[newServiceOperation.length - 1] = inp;
                    mServiceOperation = newServiceOperation;
               } else if (index >= this.mServiceOperation.length){
                    ServiceOperation [ ] newServiceOperation = new
ServiceOperation[index + 1];
                    java.lang.System.arraycopy((Object)mServiceOperation, 0,
(Object)newServiceOperation, 0, this.mServiceOperation.length);
                    newServiceOperation[index] = inp;
                    mServiceOperation = newServiceOperation;
               } else {
                    this.mServiceOperation[index] = inp;
               }
          }
          public void setServiceOperation(ServiceOperation [ ] inp){
               this.mServiceOperation = inp;
          }
          public String getServiceTerms( ){
           return mServiceTerms;
          }
          public String getServiceTermsToXML( ){
               if (getServiceTerms( ) == null) return null;
               char [ ] c = getServiceTerms( ).toCharArray( );
               StringBuffer sb = new StringBuffer( );
               for (int x = 0; x < c.length; x++){
                switch(c[x]){
                case '>':
                 sb.append("&gt;");
                 break;
                case '<':
                 sb.append("&lt;");
                 break;
                case '&':
                 sb.append("&amp;");
                 break;
                case '"':
                 sb.append("&quot;");
```

-continued

```
                    break;
                   case '\':
                    sb.append("&quot;");
                    break;
                   default:
                    if (Character.isDefined(c[x]))
                     sb.append(c[x]);
                   }
                  }
                  return sb.toString( );
          }
          public void setServiceTerms(String inp){
               this.mServiceTerms = inp;
          }
          public void serviceTermsFromXML(String n){
               setServiceTerms(n);
          }
}
package com.veo.xdk.dev.schema.test.blib;
import com.veo.vsp.doclet.meta.Document;
public class ServiceOperation extends Document{
          public static final String DOC_TYPE = "service.operation";
           String mServiceOperationName;.
           String mServiceOperationLocation;
           String mServiceOperationInput;
           String mServiceOperationOutput;
          public ServiceOperation( ){
               super(DOC_TYPE);
          mServiceOperationName = new String( );
          mServiceOperationLocation = new String( );
          mServiceOperationInput = new String( );
          mServiceOperationOutput = new String( );
          }
          public ServiceOperation(String doc_type){
               super(doc_type);
          mServiceOperationName = new String( );
          mServiceOperationLocation = new String( );
          mServiceOperationInput = new String( );
          mServiceOperationOutput = new String( );
          }
          static public ServiceOperation initServiceOperation(String
iServiceOperationName,String iServiceOperationLocation,String iServiceOperationInput,String
iServiceOperationOutput){
                  ServiceOperation obj = new ServiceOperation( );
                  obj.initializeAll(iServiceOperationName, iServiceOperationLocation,
iServiceOperationInput, iServiceOperationOutput);
                  return obj;
          }
          public void initializeAll(String iServiceOperationName,String
iServiceOperationLocation,String iServiceOperationInput,String iServiceOperationOutput){
                  mServiceOperationName = iServiceOperationName;
                  mServiceOperationLocation = iServiceOperationLocation;
                  mServiceOperationInput = iServiceOperationInput;
                  mServiceOperationOutput = iServiceOperationOutput;
          }
          public String getServiceOperationName( ){
           return mServiceOperationName;
          }
          public String getServiceOperationNameToXML( ){
               if (getServiceOperationName( ) == null) return null;
               char [ ] c = getServiceOperationName( ).toCharArray( );
               StringBuffer sb = new StringBuffer( );
               for (int x = 0; x < c.length; x++){
                switch(c[x]){
                case '>':
                 sb.append("&gt;");
                 break;
                case '<':
                 sb.append("&lt;");
                 break;
                case '&':
                 sb.append("&amp;");
                 break;
                case '"':
                 sb. append("&quot;");
                 break;
                case '\':
                 sb. append("&quot;");
                 break;
```

-continued

```
        default:
          if (Character.isDefined(c[x]))
            sb.append(c[x]);
        }
      }
      return sb.toString( );
}
public void setServiceOperationName(String inp){
      this.mServiceOperationName = inp;
}
public void serviceOperationNameFromXML(String n){
      setServiceOperationName(n);
}
public String getServiceOperationLocation( ){
 return mServiceOperationLocation;
}
public String getServiceOperationLocationToXML( ){
      if (getServiceOperationLocation( ) == null) return null;
      char [ ] c = getServiceOperationLocation( ).toCharArray( );
      StringBuffer sb = new StringBuffer( );
      for (int x = 0; x < c.length; x++){
        switch(c[x]){
        case '>':
          sb.append("&gt;");
          break;
        case '<':
          sb.append("&lt;");
          break;
        case '&':
          sb.append("&amp;");
          break;
        case '"':
          sb.append("&quot;");
          break;
        case '\'':
          sb. append("&quot;");
          break;
        default:
          if (Character.isDefined(c[x]))
            sb.append(c[x]);
        }
      }
      return sb.toString( );
}
public void setServiceOperationLocation(String inp){
      this.mServiceOperationLocation = inp;
}
public void serviceOperationLocationFromXML(String n){
      setServiceOperationLocation(n);
}
public String getServiceOperationInput( ){
 return mServiceOperationInput;
}
public String getServiceOperationInputToXML( ){
      if (getServiceOperationInput( ) == null) return null;
      char [ ] c = getServiceOperationInput( ).toCharArray( );
      StringBuffer sb = new StringBuffer( );
      for (int x = 0; x < c.length; x++){
        switch(c[x]){
        case '>':
          sb.append("&gt;");
          break;
        case '<':
          sb.append("&lt;");
          break;
        case '&':
          sb.append("&amp;");
          break;
        case '"':
          sb. append("quot;");
          break;
        case '\'':
          sb.append("&quot;");
          break;
        default:
          if (Character.isDefined(c[x]))
            sb.append(c[x]);
        }
      }
```

-continued

```
            return sb.toString( );
        }
        public void setServiceOperationInput(String inp){
            this.mServiceOperationInput = inp;
        }
        public void serviceOperationInputFromXML(String n){
            setServiceOperationInput(n);
        }
        public String getServiceOperationOutput( ){
          return mServiceOperationOutput;
        }
        public String getServiceOperationOutputToXML( ){
            if (getServiceOperationOutput( ) = null) return null;
            char [ ] c = getServiceOperationOutput( ).toCharArray( );
            StringBuffer sb = new StringBuffer( );
            for (int x = 0; x < c.length; x++){
              switch(c[x]){
              case '>':
                sb.append("&gt;");
                break;
              case '<':
                sb.append("&lt;");
                break;
              case '&':
                sb.append("&amp;");
                break;
              case '"':
                sb.append("&quot;");
                break;
              case '\':
                sb.append("&quot;");
                break;
              default:
                if (Character.isDefined(c[x]))
                  sb.append(c[x]);
              }
            }
            return sb.toString( );
        }
        public void setServiceOperationOutput(String inp){
            this.mServiceOperationOutput = inp;
        }
        public void serviceOperationOutputFromXML(String n){
            setServiceOperationOutput(n);
        }
}
package com.veo.xdk.dev.schema.test.blib;
import com.veo.vsp.doclet.meta.Document;
public class ServiceSet extends Document {
        public static final String DOC_TYPE = "service.set";
        Service [ ] mService;
        public ServiceSet( ){
            super(DOC_TYPE);
            this.mService = new Service[0];
        }
        public ServiceSet(String doc_type){
            super(doc_type);
            this.mService = new Service[0];
        }
        static public ServiceSet initServiceSet(Service [ ] iService){
            ServiceSet obj = new ServiceSet( );
            obj.initializeAll(iService);
            return obj;
        }
        public void initializeAll(Service [ ] iService){
            mService = iService;
        }
        public Service [ ] getService( ){
          return mService;
        }
        public Service getService(int index){
            if (this.mService == null)
              return null;
            if (index >= this.mService.length)
              return null;
            if (index < 0 && -index> this.mService.length)
              return null;
            if (index >= 0) return this.mService[index];
            return this.mService[this.mService.length + index];
```

-continued

```
        }
        public void setService(int index, Service inp){
          if (this.mService = null) {
            if (index < 0) {
             this.mService = new Service[1];
             this.mService[0] = inp;
            } else {
             this.mService = new Service[index + 1];
             this.mService[index] = inp;
            }
          } else if (index < 0) {
                Service [ ] newService = new Service[this.mService.length + 1];
                java.lang.System.arraycopy((Object)mService, 0, (Object)newService,
    0, this.mService.length);
                newService[newService.length - 1] = inp;
                mService = newService;
            } else if (index >= this.mService.length){
                Service [ ] newService = new Service[index + 1];
                java.lang.System.arraycopy((Object)mService, 0, (Object)newService,
    0, this.mService.length);
                newService[index] = inp;
                mService = newService;
            } else {
                this.mService[index] = inp;
        }
    }
    public void setService(Service [ ] inp){
        this.mService inp;
    }
}
```

In addition to the JAVA beans set forth above, transformation code is produced for translating from JAVA to XML and XML to JAVA as set forth below:

```
    Java to XML
<!.DOCTYPE tree SYSTEM "tree.dtd">
<tree source = "null" pass-through = "false">
<before>
 <vardef name = "attribute.def">
  <element source = "ATTRIBUTE" class = "NAME" type = "5" position = "-2">
   <parse>
    <data class = "java.lang.String" position = "-2"/>
   </parse>
  </element>
 </vardef>
 <vardef name = "pcdata.def">
  <element source = "PCDATA" class = "NAME" type = "4"position = "-2">
   <parse>
    <data class = "999" type = "6" position = "-2"/>
   </parse>
  </element>
 </vardef>
 <vardef name = "content.def">
  <element source = "PCDATA">
   <parse>
    <data class = "999" type = "6" position = "-2"/>
   </parse>
  </element>
 </vardef>
 <vardef name = "ServiceSet.var">
 <element source = "com.veo.xdk.dev.schema.test.blib.ServiceSet" class = "service.set" type = "4"
position = "-2">
   <parse>
    <callvar name = "Service.var">
   </parse>
  </element>
 </vardef>
 <vardef name = "PrototypeService.var">
 <element source = "com.veo.xdk.dev.schema.test.blib.PrototypeService" class =
"prototype.service" type = "4" position = "-2">
  <parse>
   <callvar name = "pcdata.def" parms = "setSource ServiceNameToXML setGenerator
service.name"/>
```

-continued

```
    <callvar name = "pcdata.def" parms = "setSource ServiceTermsToXML setGenerator
service.terms"/>
    <callvar name = "pcdata.def" parms = "setSource ServiceLocationToXML setGenerator
service.location"/>
    <callvar name = "ServiceOperation.var"/>
  </parse>
  </element>
  </vardef>
  <vardef name = "Service.var">
  <element source = "com.veo.xdk.dev.schema.test.blib.Service" class = "service" type = "8"
position = "0">
   <parse>
    <callvar name = "pcdata.def" parms = "setSource ServiceNameToXML setGenerator
service.name"/>
    <callvar name = "pcdata.def" parms = "setSource ServiceLocationToXML setGenerator
service.location"/>
    <callvar name = "ServiceOperation.var">
    <callvar name = "pcdata.def" parms = "setSource ServiceTermsToXML setGenerator
service.terms"/>
   </parse>
  </element>
  </vardef>
  <vardef name = "ServiceOperation.var"/>
  <element source = "com.veo.xdk.dev.schema.test.blib.ServiceOperation" class =
"service.operation" type = "4" position = "-2">
   <parse>
    <callvar name = "pcdata.def" parms = "setSource ServiceOperationNameToXML setGenerator
service.operation.name"/>
    <callvar name = "pcdata.def" parms = "setSource ServiceOperationLocationToXML
setGenerator service.operation.location"/>
    <callvar name = "pcdata.def" parms = "setSource ServiceOperationInputToXML setGenerator
service.operation.input"/>
    <callvar name = "pcdata.def" parms = "setSource ServiceOperationOutputToXML
setGenerator service.operation.output"/>
   </parse>
  </element>
  </vardef>
</before>
<parse>
<callvar name = "ServiceSet.var"/>
<callvar name = "PrototypeService.var"/>
<callvar name = "Service.var"/>
<callvar name = "ServiceOperation.var"/>
</parse>
</tree>
XML to Java
<!DOCTYPE tree SYSTEM "tree.dtd">
<tree source = "null" pass-through = "false">
<before>
<vardef name = "business.var">
 <element source = "business"
       class = "com.veo.xdk.dev.schema.test.blib.Business"
       type = "7" setter = "setBusiness">
  <before>
   <onattribute name = "business.number">
    <actions>
     <callmeth name = "businessNumberFromXML">
      <parms>
       <getattr name = "business.number"/>
      </parms>
     </callmeth>
    </actions>
   </onattribute>
  <before>
  <parse>
   <callvar name = "party.name.var" parms = "setPosition -1"/>
   <callvar name = "address.set.var"/>
  </parse>
 </element>
</vardef>
<vardef name = "party.name.var">
 <element source = "party.name" setter = "partyNameFromXML" position = "-1" class =
"java.lang.String">
  <parse>
   <data class = "java.lang.String" position = "0"/>
  </parse>
 </element>
</vardef>
<vardef name = "city.var">
```

-continued

```
<element source = "city" setter = "cityFromXML" position = "-1" class = "java.lang.String">
  <parse>
    <data class = "java.lang.String" position = "0"/>
  </parse>
 </element>
</vardef>
<vardef name = "internet.var">
 <element source = "internet" setter = "internetFromXML" position = "-1" class =
"java.lang.String">
  <parse>
    <data class = "java.lang.String" position = "0"/>
  </parse>
 </element>
</vardef>
<vardef name = "country.var">
 <element source = "country" setter = "countryFromXML" position = "1" class =
"java.lang.String">
  <parse>
    <data class = "java.lang.String" position = "0"/>
  /<parse>
 </element>
</vardef>
<vardef name = "state.var">
 <element source = "state" setter = "stateFromXML" position = "-1" class = "java.lang.String">
  <parse>
    <data class = "java.lang.String" position = "0"/>
  </parse>
 </element>
</vardef>
<vardef name = "email.var">
 <element source = "email" setter "emailFromXML" position = "-1" class = "java.lang.String">
  <parse>
    <data class = "java.lang.String" position = "0"/>
  </parse>
 </element>
</vardef>
<vardef name = "address.physical.var">
 <element source = "address.physical"
       class = "com.veo.xdk.dev.schema.test.blib.AddressPhysical"
       type = "7" setter = "setAddressPhysical">
  <before>
  </before>
  <parse>
   <callvar name = "streetvar" parms = "setPosition -1"/>
   <callvar name = "city.var" parms = "setPosition -1"/>
   <callvar name = "state.var" parms = "setPosition -1"/>
   <callvar name = "postcode.var" parms = "setPosition -1"/>
   <callvar name = "country.var" parms = "setPosition -1"/>
  </parse>
 </element>
</vardef>
<vardef name = "telephone.var">
 <element source = "telephone" setter = "telephoneFromXML" position = "-1" class =
"java.lang.String">
  <parse>
    <data class = "java.lang.String" position = "0"/>
  </parse>
 </element>
</vardef>
<vardef name = "person.var">
 <element source = "person"
       class = "com.veo.xdk.dev.schema.test.blib.Person"
       type = "7" setter = "setPerson">
  <before>
   <onattribute name = "SSN">
    <actions>
     <callmeth name = "sSNFromXML">
      <parms>
       <getattr name = "SSN"/>
      </parms>
     </callmeth>
    </actions>
   </onattribute>
  </before>
  <parse>
   <callvar name = "party.name.var" parms = "setPosition -1"/>
   <callvar name = "address.set.var"/>
  </parse>
 </element>
```

-continued

```
</vardef>
<vardef name = "fax.var">
 <element source = "fax" setter = "faxFromXML" position = "-1" class = "java.lang.String">
  <parse>
   <data class = "java.lang.String" position = "0"/>
  <parse>
 <element>
</vardef>
<vardef name = "street.var">
 <element source = "street" setter = "streetFromXML" position = "-1" class = "java.lang.String">
  <parse>
   <data class = "java.lang.String" position = "0"/>
  </parse>
 </element>
</vardef>
<vardef name = "address.set.var">
 <element source = "address.set"
       class = "com.veo.xdk.dev.schema.test.blib.AddressSet"
       type = "7" setter = "setAddressSet">
  <before>
  </before>
  <parse>
   <callvar name = "address.physical.var">
   <callvar name = "telephone.var" parms = "setPosition -1"/>
   <callvar name = "fax.var" parms = "setPosition -1"/>
   <callvar name = "email.var" parms = "setPosition -1"/>
   <callvar name = "internet.var" parms = "setPosition "-1"/>
  </parse>
 </element>
</vardef>
<vardef name = "postcode.var">
 <element source = "postcode" setter = "postcodeFromXML" position = "-1" class =
"java.lang.String">
  <parse>
   <data class = "java.lang.String" position = "0"/>
  </parse>
 </element>
</vardef>
<vardef name = "market.participant.var">
 <element source = "market.participant"
       class = "com.veo.xdk.dev.schema.test.blib.MarketParticipant"
       type = "7" position = "0"/>
  <before>
  </before>
  <parse>
   <callvar name = "business.var"/>
   <callvar name = "person.var"/>
  </parse>
 </element>
</vardef>
</before>
<parse>
<callvar name = "business.var"/>
<callvar name = "party.name.var"/>
<callvar name = "city.var"/>
<callvar name = "internet.var"/>
<callvar name = "country.var"/>
<callvar name = "state.var"/>
<callvar name = "email.var"/>
<callvar name = "address.physical.var"/>
<callvar name = "telephone.var"/>
<callvar name = "person.var"/>
<callvar name = "fax.var"/>
<callvar name = "street.var"/>
<callvar name = "address.set.var"/>
<callvar name = "postcode.var"/>
<callvar name = "market.participant.var"/>
</parse>
</tree>
```

Makefiles:

```
#
# this makefile was generated by bic version 0.0. 05/02/1998
#
#
#
#
# get the package name from the package argument passed to SchemaGen
PACKAGE_NAME = com/veo/xdk/dev/schema/test/blib
JAVA_SOURCES    +=          \
        MarketParticipant.java \
        Business.java \
        Person.java \
        Party.java \
        AddressPhysical.java \
        AddressSet.java \
MAKEFILE_MASTER_DIR = xxx
include $(MAKEFILE_MASTER_DIR)/Makefile.master
all:: $(JAVA_CLASSES)
#
# this makefile was generated by bic version 0.0. 05/02/1998
#
#
#
#
```

-continued

Makefiles:

```
#
# this makefile was generated by bic version 0.0. 05/02/1998
#
#
#
#
# get the package name from the package argument passed to SchemaGen
PACKAGE_NAME = com/veo/xdk/dev/schema/test/blib
JAVA_SOURCES    +=          \
# get the package name from the package argument passed to SchemaGen
PACKAGE_NAME = com/veo/xdk/dev/schema/test/blib
JAVA_SOURCES    +=          \
        ServiceSet.java \
        PrototypeService.java \
        Service.java \
        ServiceOperation.java \
MAKEFILE_MASTER_DIR = xxx
include $(MAKEFILE_MASTER_DIR)/Makefile.master
all:: $(JAVA_CLASSES)
```

Finally, the XML document instances generated at run time according to the model above for one example follows:

```
<!DOCTYPE market.participant SYSTEM "market.participant.dtd">
<market.participant>
<business business.number="1234567890">
<party.name>IBM</party.name>
<address.set>
<address.physical>
<street>1 IBM Way</street>
<city>Palo Alto</city>
<state>CA</state>
<postcode>94304</postcode>
<country>USA</country>
</address.physical>
<telephone>123 456-7890</telephone>
<fax>123 456 0987</fax>
<email>ibmec@ibm.com</email>
</address.set>
</business>
</market.participant>
<!DOCTYPE service SYSTEM "service.dtd">
<service.set>
<service>
<service.name>Order Service</service.name>
<service.location>www.ibm.com/order</service.location>
<service.operation>
<service.operation.name>Submit Order<service.operation.name>
<service.operation.location>www.ibm.com/order/submit</service.location>
<service.operation.input>urn:x-ibm:services:order:operations:po.dtd</service.operation.input>
<service.operation.outut>urn:x-
ibm:services:order:operations:poack.dtd</service.operation.output>
</service.operation>
<service.operation>
<service.operation.name>Track Order</service.operation.name>
<service.operation.location>www.ibm.com/order/track</service.location>
<service.operation.input>urn:x-
ibm:services:order:operations:track.irequest.dtd</service.operation.input>
<service.operation.output>urn:x-
ibm:services:order:operations:track.iresponse.dtd</service.operation.output>
</service.operation>
</service>
</service.set>
```

Using the tools along with a BID composer application, which provides a drag, drop and forms editing user interface, a developer is able to create a business interface definition and to produce a well formed, valid business interface definition in the form of an XML document. Thus, the

example run time instance is a business interface definition for an ordering service for IBM to be used by Ingram Micro and others to order laptop computers from IBM. (There is no relationship between the applicant and IBM or Ingram Micro). Utilizing these processes, a user is able to build a system that allows for programming of a business interface using the documents defined according to the present invention.

The role of CBL and the BID processor of the present invention in an XML/JAVA environment can be further understood by the following explanation of the processing of a Purchase Order.

Company A defines its Purchase Order document type using a visual programming environment that contains a library of CBL DTDs and modules, all defined using common business language elements so that they contain data type and other interpretation information. Company A's PO might just involve minor customizations to a more generic "transaction document" specification that comes with the CBL library, or it might be built from the ground up from CBL modules for address, date and time, currency, etc.

The documentation for the generic "transaction document" specification (such as the transact.dtd set out above) typifies the manner in which CBL specifications are built from modules and are interlinked with other CBL DTDs.

A compiler takes the purchase order definition and generates several different target forms. All of these target forms can be derived through "tree to tree" transformations of the original specification. The most important for this example are:

(a) the XML DTD for the purchase order.

(b) a JAVA Bean that encapsulates the data structures for a purchase order (the JAVA classes, arguments, datatypes, methods, and exception structures are created that correspond to information in the Schema definition of the purchase order).

(c) A "marshaling" program that converts purchase orders that conform to the Purchase Order DTD into a Purchase Order JAVA Bean or loads them into a database, or creates HTML (or an XSL style sheet) for displaying purchase orders in a browser.

(d) An "unmarshaling" program that extracts the data values from Purchase Order JAVA Beans and converts them into an XML document that conforms to the Purchase Order DTD.

Now, back to the scenario. A purchasing application generates a Purchase Order that conforms to the DTD specified as the service interface for a supplier who accepts purchase orders.

The parser uses the purchase order DTD to decompose the purchase order instance into a stream of information about the elements and attribute values it contains. These "property sets" are then transformed into corresponding JAVA event objects by wrapping them with JAVA code. This transformation in effect treats the pieces of marked-up XML document as instructions in a custom programming language whose grammar is defined by the DTD. These JAVA events can now be processed by the marshaling applications generated by the compiler to "load" JAVA Bean data structures.

Turning the XML document into a set of events for JAVA applications to process, is unlike the normal model of parsing in which the parser output is maintained as an internal data structure and processing does not begin until parsing completes. The event based processing, in response to the BID definitions, is the key to enabling the much richer functionality of the processor because it allows concurrent document application processing to begin as soon as the first event is emitted.

JAVA programs that "listen for" events of various types are generated from the Schema definition of those events. These listeners are programs created to carry out the business logic associated with the XML definitions in the CBL; for example, associated with an "address" element may be code that validates the postal code by checking a database. These listeners "subscribe" to events by registering with the document router, which directs the relevant events to all the subscribers who are interested in them.

This publish and subscribe architecture means that new listener programs can be added without knowledge by or impact on existing ones. Each listener has a queue into which the router directs its events, which enables multiple listeners can handle events in parallel at their own pace.

For the example purchase order here, there might be listeners for:

- the purchase order, which would connect it to an order entry program,
- product descriptions, which might check inventory,
- address information, which could check Fed Ex or other service for delivery availability,
- buyer information, which could check order history (for creditworthiness, or to offer a promotion, or similar processing based on knowing who the customer is).

Complex listeners can be created as configurations of primitive ones (e.g., a purchase order listener may contain and invoke these listeners here, or they may be invoked on their own).

FIG. **11** illustrates the market maker node in the network of FIG. 1. The market maker node includes the basic structures of the system of FIG. **3**, including a network interface **1101**, a document parser **1102**, a document to host and host to document translator **1103**, and a front end **1104**, referred to as a router in this example. The market maker module **1105** in this example includes a set of business interface definitions, or other identifiers sufficient to support the market maker function, for participants in the market, a CBL repository, and a compiler all serving the participants in the market. The router **1104** includes a participant registry and document filters which respond to the events generated at the output of the translator and by the parser to route incoming documents according to the participant registry and according to the element and attribute filters amongst the listeners to the XML event generators. Thus, certain participants in the market may register to receive documents that meet prespecified parameters. For example, input documents according to a particular DTD, and including an attribute such as numbers of products to be purchased greater than a threshold, or such as a maximum price of a document request to be purchased, can be used to filter documents at the router **1104**. Only such documents as match the information registered in the participant registry at the router **1104** are then passed on to the registered participant.

The router **1104** may also serve local host services **1105** and **1106**, and as such act as a participant in the market as well as the market maker. Typically, documents that are received by the router **1104** are traversed to determine the destinations to which such documents should be routed, there again passed back through the translator **1103**, if necessary, and out the network interface **1101** to the respective destinations.

The market maker is a server that binds together a set of internal and external business services to create a virtual enterprise or trading community. The server parses incoming documents and invokes the appropriate services by, for example, handing off a request for product data to a catalog server or forwarding a purchase order to an ERP system. The server also handles translation tasks, mapping the information from a company's XML documents onto document formats used by trading partners and into data formats required by its legacy systems.

With respect to the service definition above, when a company submits a purchase order, the XML parser in the server uses the purchase order DTD to transform the purchase order instance into a stream of information events. These events are then routed to any application that is programmed to handle events of a given type; in some cases, the information is forwarded over the Internet to a different business entirely. In the purchase order example, several applications may act on information coming from the parser:

An order entry program processes the purchase order as a complete message;

An ERP system checks inventory for the products described in the purchase order;

A customer database verifies or updates the customer's address;

A shipping company uses the address information to schedule a delivery

A bank uses the credit card information to authorize the transaction.

Trading partners need only agree on the structure, content, and sequencing of the business documents they exchange, not on the details of APIs. How a document is processed and what actions result is strictly up to the business providing the service. This elevates integration from the system level to the business level. It enables a business to present a clean and stable interface to its business partners despite changes in its internal technology implementation, organization, or processes.

FIGS. **12, 13** and **14** illustrate processes executed at a market maker node in the system of FIG. **11**. In FIG. **12**, an input document is received at the network interface from an originating participant node (step **1200**). The document is parsed (step **1201**). The document is translated to the format of the host, for example XML to JAVA (step **1202**). The host formatted events and objects are then passed to the router service (step **1203**). The services registered to accept the document according to the document type and content of the document are identified (step **1204**). The document or a portion of the document is passed to the identified services (step **1205**). As service is performed in response to the document content (step **1206**). The output data of the service is produced (step **1207**). The output is converted to the document format, for example from a JAVA format to an XML format (step **1208**). Finally, the output document is sent to a participant node (step **1209**).

The registration service is one such function which is managed by the router. Thus, a market participant document is accepted at the network interface as shown in FIG. **13** (step **1300**). The market participant document is stored in the business interface definition repository (step **1301**) for the market maker node. In addition, the document is parsed (step **1302**). The parsed document is translated into the format of the host (step **1303**). Next, the document is passed to the router service (step **1304**). The router service includes a listener which identifies the registration service as the destination of the document according to the document type and content (step **1305**). The document or elements of the document are passed to the registration service (step **1306**). In the registration service, the needed service specifications are retrieved according to the business interface definition (step **1307**). If the service specifications are gathered, at step **1308**, the router service filters are set according to the business interface definition and the service specifications (step **1309**). Registration acknowledgment data is produced (**1310**). The registration acknowledgment data is converted to a document format (step **1311**). Finally, the acknowledgment document is sent to the participant node indicating to the participant that is successfully registered with the market maker (step **1312**).

The process at step **1307** of gathering needed service specifications is illustrated for one example in FIG. **14**. This process begins by locating a service business interface definition supported by the market participant (step **1400**). The service definition is retrieved, for example by an E-mail transaction or web access to repository node (step **1401**). The service specification is stored in the BID repository (step **1402**). The service business interface definition document is parsed (step **1403**). The parsed document is translated into the format of the host (step **1404**). Host objects are passed to the router service (step **1405**). The registration service is identified according to the document type and content (step **1406**). Finally, the information in the service business interface definition document is passed to the registration service (step **1407**) for use according to the process of FIG. **13**.

FIG. **15** illustrates the processor, components and sequence of processing of incoming data at market maker node according to the present invention. The market maker node includes a communication agent **1500** at the network interface. The communication agent is coupled with an XML parser **1501** which supplies events to an XML processor **1502**. The XML processor supplies events to a document router. The document router feeds a document service **1504** that provides an interface for supplying the received documents to the enterprise solution software **1505** in the host system. The communication agent **1500** is an Internet interface which includes appropriate protocol stacks supporting such protocols as HTTP, SMTP, FTP, or other protocols. Thus, the incoming data could come in an XML syntax, an ASCII data syntax or other syntax as suits a particular communication channel. All the documents received in non-XML syntaxes are translated into XML and passed the XML parser. A translation table **1506** is used to support the translation from non-XML form into XML form.

The converted documents are supplied to the parser **1501**. The XML parser parses the received XML document according to the document type definition which matches it. If an error is found, then the parser sends the document back to the communication agent **1500**. A business interface definition compiler BIDC **1507** acts as a compiler for business interface definition data. The DTD file for the XML parser, JAVA beans corresponding to the DTD file, and translation rules for translating DTD files to JAVA beans are created by compiling the BID data. An XML instance is translated to JAVA instance by referring to these tools. Thus the BID compiler **1507** stores the DTD documents **1508** and produces JAVA documents which correspond **1509**. The XML documents are passed to the processor **1502** which translates them into the JAVA format. In a preferred system, JAVA documents which have the same status as the document type definitions received in the XML format are produced. The JAVA beans are passed to the document router **1503**. The document router **1503** receives the JAVA beans and passes the received class to the appropriate document service using a registry program, for example using the event listener architecture described above. The document service **1504** which receives the document in the form of JAVA beans from the router **1503** acts as the interface to the enterprise solution software. This includes a registry service **1510** by which listeners to XML events are coupled with the incoming data streams, and a service manager **1511** to manage the routing of the incoming documents to the appropriate services. The document service manager **1511** provides for administration of the registry service and for maintaining document consistency and the like.

The document service communicates with the back end system using any proprietary API, or using such more common forms as the CORBA/COM interface or other architectures.

FIG. 16 provides a heuristic diagram of the market maker and market participant structures according to the present invention. Thus, the electronic commerce market according to the present invention can be logically organized as set forth in FIG. 16. At the top of the organization, a market maker node **1600** is established. The market maker node includes resources that establish a marketplace **1601**. Such resources include a market registry service and the like. Businesses **1602** register in the marketplace **1601** by publishing a business interface definition. The business interface definition defines the services **1603** for commercial transactions in which the businesses will participate. The transactions **1604** and services **1603** use documents **1605** to define the inputs and outputs, and outline the commercial relationship between participants in the transaction. The documents have content **1606** which carries the particulars of each transaction. The manner in which the content is processed by the participants in the market, and by the market maker is completely independent of the document based electronic commerce network which is established according to the present invention. Overall, a robust, scalable, intuitive structure is presented for enabling electronic commerce on communication networks is provided.

Thus, the present invention in an exemplary system provides a platform based on the XML processor and uses XML documents as the interface between loosely coupled business systems. The documents are transferred between businesses and processed by participant nodes before entering the company business system. Thus the platform enables electronic commerce applications between businesses where each business system operates using different internal commerce platforms, processes and semantics, by specifying a common set of business documents and forms.

According to the present invention, virtual enterprises are created by interconnecting business systems and service, are primarily defined in terms of the documents (XML-encoded) that businesses accept and generate:

"if you send me a request for a catalog, I will send you a catalog:

"if you send me a purchase order and I can accept it, I will send you an invoice".

The foregoing description of a preferred embodiment of the invention has been presented for purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Obviously, many modifications and variations will be apparent to practitioners skilled in this art. It is intended that the scope of the invention be defined by the following claims and their equivalents.

What is claimed is:

1. A method for managing transactions among nodes in a network including a plurality of nodes which execute processes involved in the transactions, comprising:

storing machine-readable specifications of a plurality of participant interfaces, the participant interfaces identifying transactions, the respective transactions being identified by definitions of input documents, and definitions of output documents, the definitions of the input and output documents comprising respective descriptions of sets of storage units and logical structures for the sets of storage units;

receiving data comprising a document through a communication network;

parsing the document according to the specifications to identify an input document and one or more transactions which accept the identified input document;

providing at least a portion of the input document in a machine-readable format to transaction processes associated with the one or more identified transactions.

2. The method of claim **1**, including:

providing a repository storing a library of logical structures, schematic maps for logic structures, and definitions of documents comprising logic structures used to build participant interface descriptions.

3. The method of claim **2**, including providing access to the repository through the communication network to other nodes in the network.

4. The method of claim **1**, wherein the machine-readable specification includes documents compliant with a definition of a participant interface document including logical structures for storing an identifier of a particular transaction, and at least one of definitions and references to definitions of input and output documents for the particular transaction.

5. The method of claim **1**, wherein the machine-readable specifications include documents compliant with a definition of a participant interface document including logical structures for storing an identifier of the participant interface, and for storing at least one of specifications and references to specifications of a set of one or more transactions supported by the participant interface.

6. The method of claim **5**, wherein the documents compliant with a definition of a participant interface document include a reference to a specification of a particular transaction, and the specification of the particular transaction includes a document including logical structures for storing at least one of definitions and references to definitions of input and output documents for the particular transaction.

7. The method of claim **1**, wherein the storage units comprise parsed data.

8. The method of claim **7**, wherein the parsed data in at least one of the input and output documents comprises:

character data encoding text characters in the one of the input and output documents, and

markup data identifying sets of storage units according to the logical structure of the one of the input and output documents.

9. The method of claim **8**, wherein at least one of the sets of storage units encodes a plurality of text characters providing a natural language word.

10. The method of claim **9**, wherein the specification includes interpretation information for at least one of the sets of storage units identified by the logical structure of at least one of the input and output documents, encoding respective definitions for sets of parsed characters.

11. The method of claim **9**, wherein the storage units comprise unparsed data.

12. The method of claim **1**, wherein the providing at least a portion of the input document in a machine-readable format to transaction processes associated with the one or more identified transactions includes executing a routing process according to a processing architecture, and including:

compiling in response to the definitions of the input and output documents in the participant interfaces, data structures corresponding to the sets of storage units and logical structures of the input and output documents compliant with the processing architecture of the transaction process, instructions executable by the system to translate the input document to the corresponding data structures.

13. The method of claim **1**, wherein the providing at least a portion of the input document in a machine-readable format to transaction processes associated with the one or more identified transactions includes executing a routing process according to a processing architecture, and including translating at least of portion of the incoming document into a format readable according to the processing architecture.

14. The method of claim 13, wherein the translating includes producing programming objects including variables and methods according to the processing architecture of the routing process.

15. The method of claim 1, wherein providing at least a portion of the input document in a machine-readable format to transaction processes associated with the one or more identified transactions, includes routing the portion of the input document to the identified transactions.

16. The method of claim 15, wherein the routing includes sending the input document on the communication network to a node executing one of the identified transactions.

17. The method of claim 1, wherein the definitions of the input and output documents comprise document type definitions compliant with a standard Extensible Markup Language XML.

18. The method of claim 17, wherein the specifications of participant interfaces comprise definitions of documents according to document type definitions compliant with a standard Extensible Markup Language XML.

19. The method of claim 1, wherein the repository includes standardized document types for use in a plurality of transactions, and wherein the definition of one of the input and output documents includes a reference to a standardized document type in the repository.

20. The method of claim 19, wherein the repository includes a standardized document type for identifying participant processes in the network.

21. The method of claim 19, including providing a repository of interpretation information for logical structures, including interpretation information identifying parameters of transactions.

22. The method of claim 1, wherein the transaction processes have respectively one of a plurality of variant transaction processing architectures, and including translating at least of portion of the incoming document into a format readable according to the variant transaction processing architecture of the respective transaction processes, and routing the translated portion to the respective transaction processes.

23. The method of claim 22, wherein the translating includes producing programming objects including variables and methods according to the variant transaction processing architecture of the respective transaction processes.

24. The method of claim 23, wherein the variant transaction processing architectures of the transaction processes comprises a process compliant with an interface description language.

25. Apparatus for managing transactions among nodes in a network including a plurality of nodes which execute processes involved in the transactions, comprising:

a network interface;

memory storing data and programs of instructions, including machine-readable specifications of a plurality of participant interfaces, the participant interfaces identifying transactions, the respective transactions being identified by definitions of input documents, and definitions of output documents, the definitions of the input and output documents comprising respective descriptions of sets of storage units and logical structures for the sets of storage units;

a data processor coupled to the memory and the network interface which executes the programs of instructions;

wherein the programs of instructions include

logic to receive data comprising a document through a network interface;

logic to parse the document according to the specifications to identify an input document and one or more transactions which accept the identified input document; and

logic to provide at least a portion of the input document in a machine-readable format to transaction processes associated with the one or more identified transactions.

26. The apparatus of claim 25, including a repository stored in memory accessible by the data processor storing a library of logical structures, schematic maps for logic structures, and definitions of documents comprising logic structures used to build participant interface descriptions.

27. The apparatus of claim 25, including logic to access a repository stored in memory through the network interface storing a library of logical structures, schematic maps for logic structures, and definitions of documents comprising logic structures used to build participant interface descriptions.

28. The apparatus of claim 25, wherein the machine-readable specification includes documents compliant with a definition of a participant interface document including logical structures for storing an identifier of a particular transaction, and at least one of definitions and references to definitions of input and output documents for the particular transaction.

29. The apparatus of claim 25, wherein the machine-readable specifications include documents compliant with a definition of a participant interface document including logical structures for storing an identifier of the participant interface, and for storing at least one of specifications and references to specifications of a set of one or more transactions supported by the participant interface.

30. The apparatus of claim 29, wherein the documents compliant with a definition of a participant interface document include a reference to a specification of a particular transaction, and the specification of the particular transaction includes a document including logical structures for storing at least one of definitions and references to definitions of input and output documents for the particular transaction.

31. The apparatus of claim 25, wherein the storage units comprise parsed data.

32. The apparatus of claim 31, wherein the parsed data in at least one of the input and output documents comprises:

character data encoding text characters in the one of the input and output documents, and

markup data identifying sets of storage units according to the logical structure of the one of the input and output documents.

33. The apparatus of claim 32, wherein at least one of the sets of storage units encodes a plurality of text characters providing a natural language word.

34. The apparatus of claim 33, wherein the specification includes interpretation information for at least one of the sets of storage units identified by the logical structure of at least one of the input and output documents, encoding respective definitions for sets of parsed characters.

35. The apparatus of claim 33, wherein the storage units comprise unparsed data.

36. The apparatus of claim 25, wherein the logic to provide at least a portion of the input document in a machine-readable format to transaction processes associated with the one or more identified transactions includes a routing process according to a processing architecture, and including:

a compiler responsive to the definitions of the input and output documents in the participant interfaces, to compile data structures corresponding to the sets of storage units and logical structures of the input and output documents compliant with the processing architecture of the transaction process, and to compile instructions

executable by the system to translate the input document to the corresponding data structures.

37. The apparatus of claim 25, wherein the logic to provide at least a portion of the input document in a machine-readable format to transaction processes associated with the one or more identified transactions includes a routing process according to a processing architecture, and including logic to translate at least of portion of the incoming document into a format readable according to the processing architecture.

38. The apparatus of claim 37, wherein the logic to translate includes producing programming objects including variables and methods according to the processing architecture of the routing process.

39. The apparatus of claim 25, wherein logic to provide at least a portion of the input document in a machine-readable format to transaction processes associated with the one or more identified transactions, includes a router to route the portion of the input document to the identified transactions.

40. The apparatus of claim 39, wherein the router includes logic to send the input document on the network interface to a node executing one of the identified transactions.

41. The apparatus of claim 25, wherein the definitions of the input and output documents comprise document type definitions compliant with a standard Extensible Markup Language XML.

42. The apparatus of claim 41, wherein the specifications of participant interfaces comprise definitions of documents according to document type definitions compliant with a standard Extensible Markup Language XML.

43. The apparatus of claim 26, wherein the repository includes standardized document types for use in a plurality of transactions, and wherein the definition of one of the input and output documents includes a reference to a standardized document type in the repository.

44. The apparatus of claim 26, wherein the repository includes a standardized document type for identifying participant processes in the network.

45. The apparatus of claim 25, wherein the transaction processes have respectively one of a plurality of variant transaction processing architectures, and including logic to translate at least of portion of the incoming document into a format readable according to the variant transaction processing architecture of the respective transaction processes, and to route the translated portion to the respective transaction processes.

46. The apparatus of claim 45, wherein the logic to translate produces programming objects including variables and methods according to the variant transaction processing architecture of the respective transaction processes.

47. The apparatus of claim 45, wherein the variant transaction processing architectures of the transaction processes comprises a process compliant with an interface description language.

* * * * *

US006052711A

United States Patent [19]

Gish

[11] Patent Number: 6,052,711

[45] Date of Patent: *Apr. 18, 2000

[54] **OBJECT-ORIENTED SYSTEM, METHOD AND ARTICLE OF MANUFACTURE FOR A CLIENT-SERVER SESSION WEB ACCESS IN AN INTERPRISE COMPUTING FRAMEWORK SYSTEM.**

[75] Inventor: **Sheri L. Gish**, Mountain View, Calif.

[73] Assignee: **Sun Microsystems, Inc.**, Mountain View, Calif.

[*] Notice: This patent issued on a continued prosecution application filed under 37 CFR 1.53(d), and is subject to the twenty year patent term provisions of 35 U.S.C. 154(a)(2).
This patent is subject to a terminal disclaimer.

[21] Appl. No.: **08/675,252**

[22] Filed: **Jul. 1, 1996**

[51] **Int. Cl.**[7] **G06F 13/14**; G06F 12/08

[52] **U.S. Cl.** **709/203**; 709/227; 709/237; 709/245; 709/201; 709/303; 709/304

[58] **Field of Search** 395/200.33, 200.31, 395/200.3, 200.45, 200.46, 200.47, 200.57, 200.67, 200.75

[56] **References Cited**

U.S. PATENT DOCUMENTS

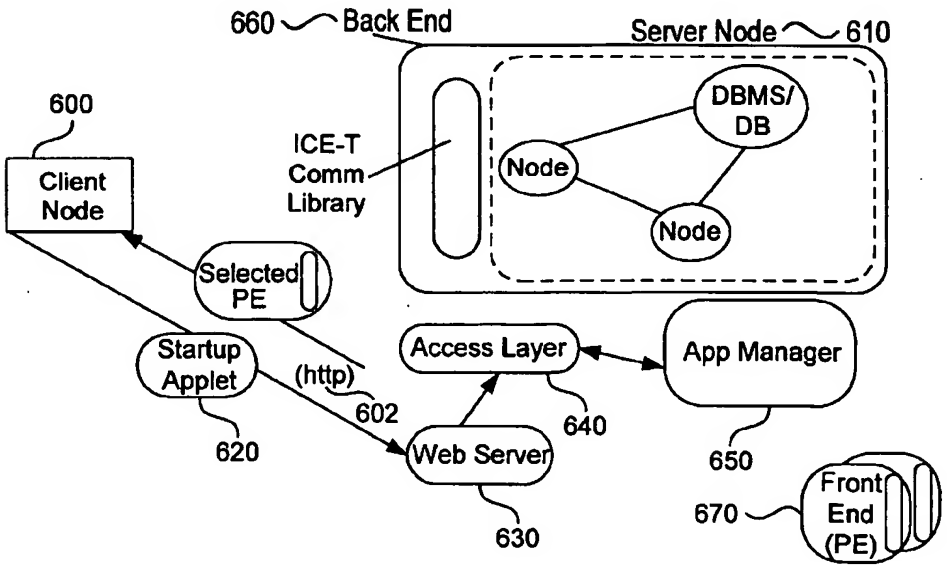
4,800,488	1/1989	Agrawal et al.	709/225
5,305,440	4/1994	Morgan et al.	709/203
5,313,635	5/1994	Ishizuka et al.	395/705
5,475,819	12/1995	Miller et al.	709/203
5,497,463	3/1996	Stein et al.	709/203
5,682,534	10/1997	Kapoor et al.	709/304
5,768,510	6/1998	Gish .	
5,848,246	12/1998	Gish .	

Primary Examiner—B. James Peikari
Attorney, Agent, or Firm—Beyer & Weaver, LLP

[57] **ABSTRACT**

An interprise computing manager in which an application is composed of a client (front end) program which communicates utilizing a network with a server (back end) program. The client and server programs are loosely coupled and exchange information using the network. The client program is composed of a User Interface (UI) and an object-oriented framework (Presentation Engine (PE) framework). The UI exchanges data messages with the framework. The framework is designed to handle two types of messages: (1) from the UI, and (2) from the server (back end) program via the network. The framework includes a component, the mediator which manages messages coming into and going out of the framework. The system includes software for a client computer, a server computer and a network for connecting the client computer to the server computer which utilize an execution framework code segment configured to couple the server computer and the client computer via the network, by a plurality of client computer code segments resident on the server, each for transmission over the network to a client computer to initiate coupling; and a plurality of server computer code segments resident on the server which execute on the server in response to initiation of coupling via the network with a particular client utilizing the transmitted client computer code segment for communicating via a particular communication protocol. Communication is initiated utilizing the network to acquire characteristics of the client from the network.

27 Claims, 17 Drawing Sheets



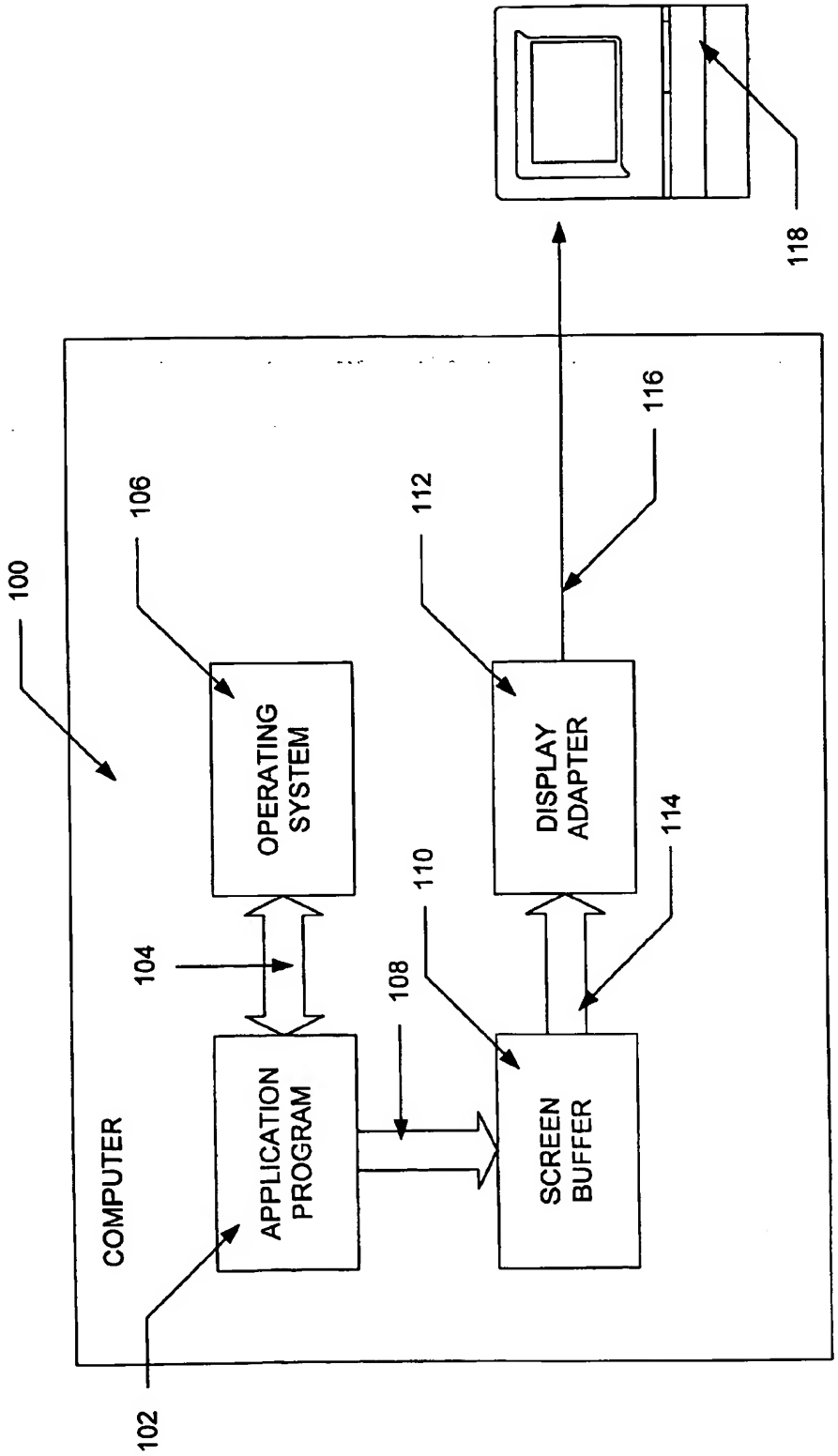


FIG. 1 (Prior Art)

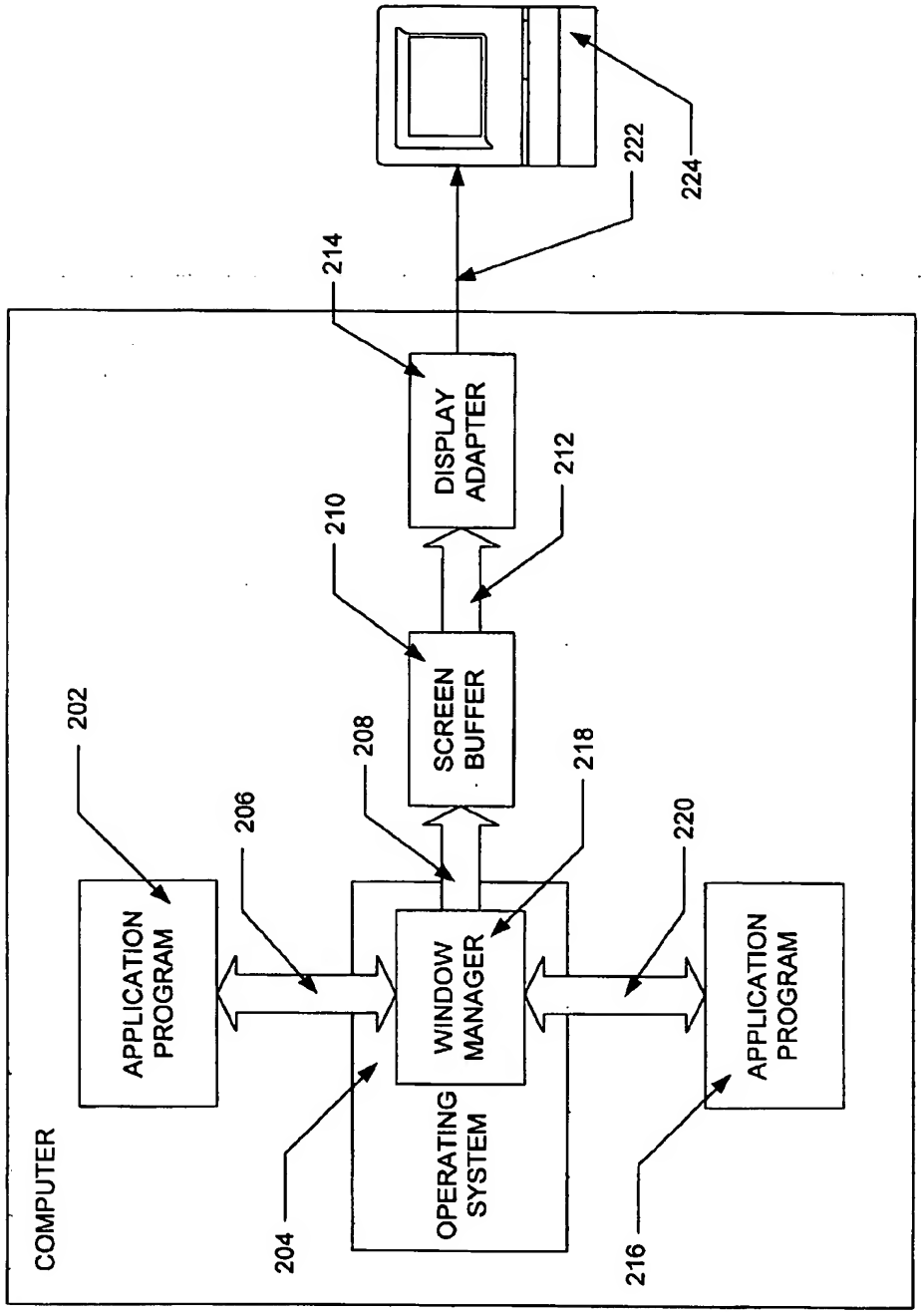


FIG. 2 (Prior Art)

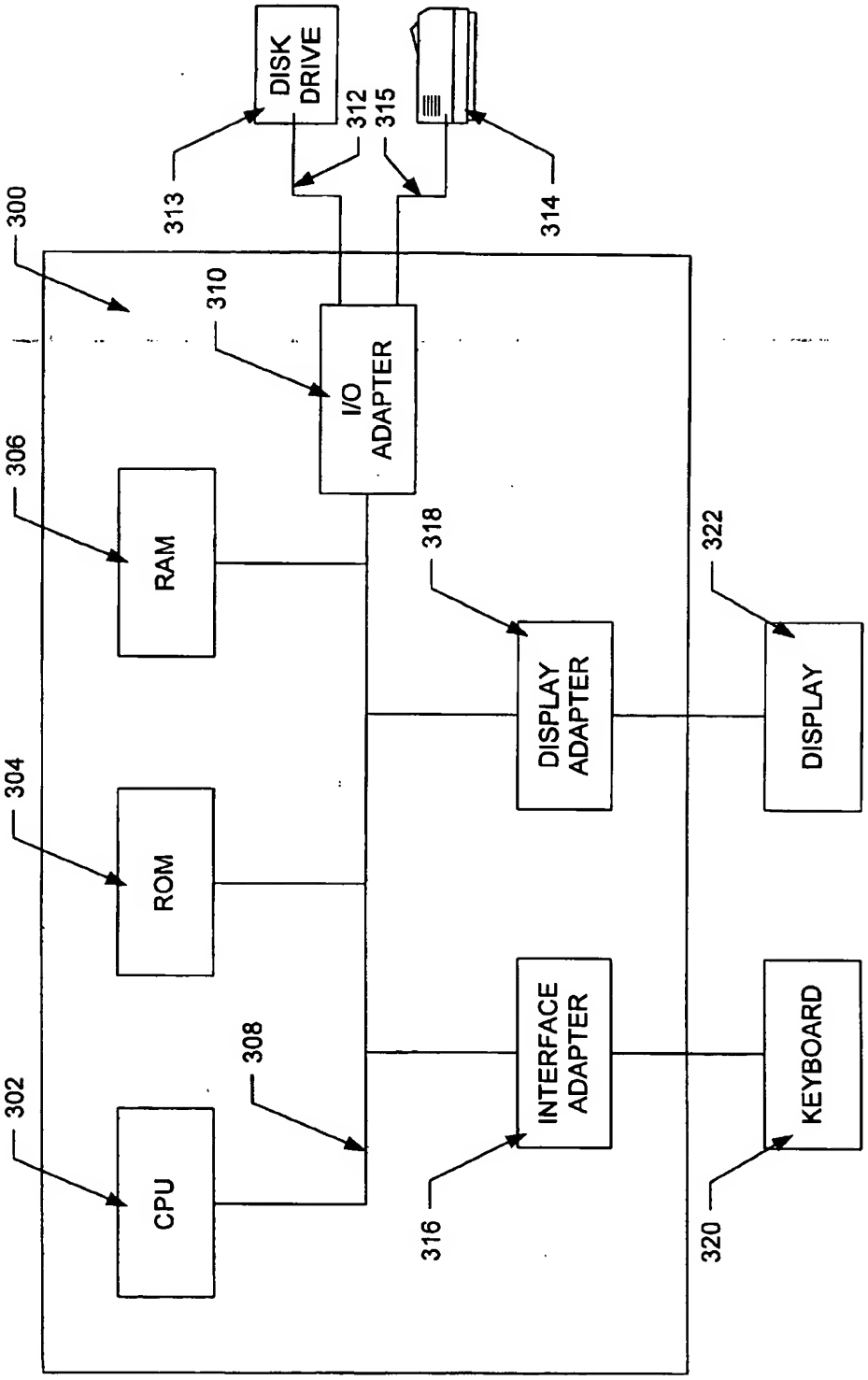

DISK DRIVE
312
315
313
314
300
310
I/O ADAPTER
306
RAM
318
322
DISPLAY ADAPTER
DISPLAY
304
ROM
INTERFACE ADAPTER
KEYBOARD
308
302
CPU
316
320

FIG. 3

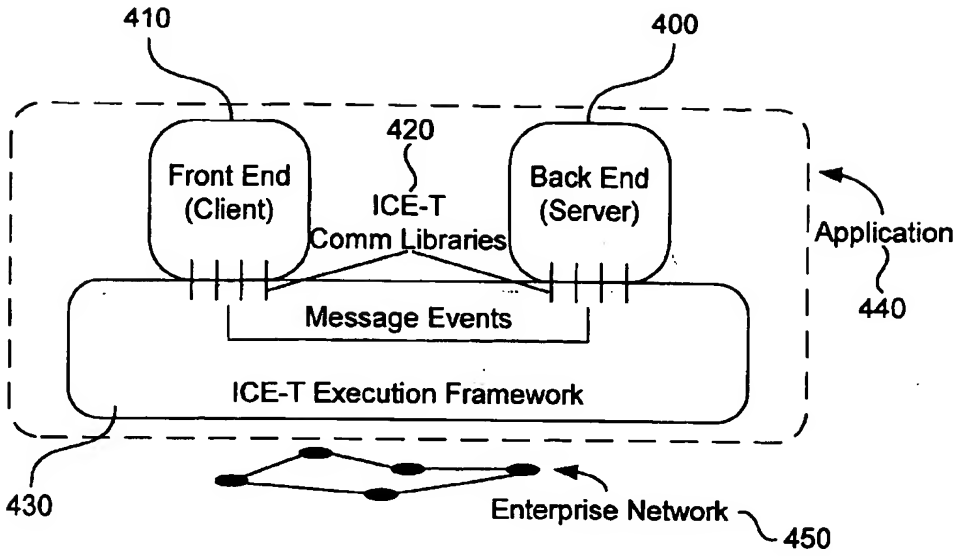

410
400
420
Front End
(Client)
ICE-T
Comm Libraries
Back End
(Server)
Application
440
Message Events
ICE-T Execution Framework
430
Enterprise Network
450

FIG. 4

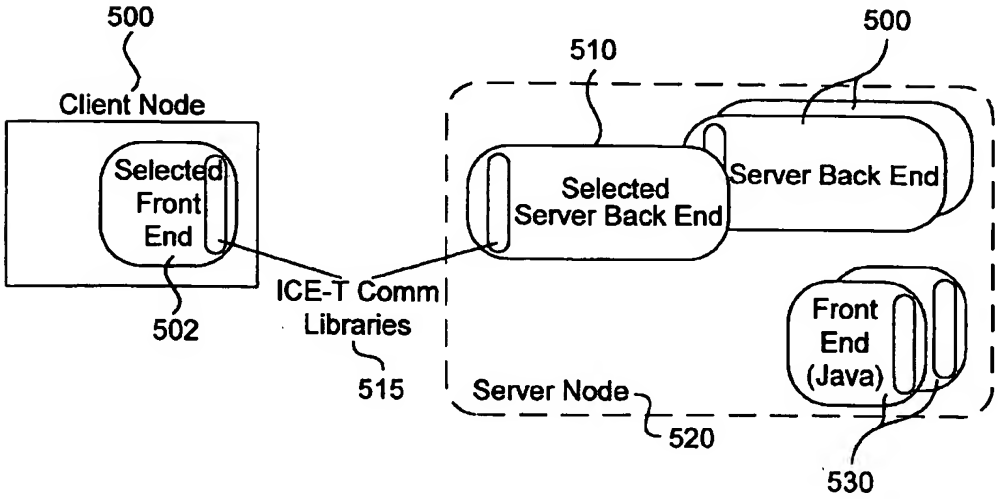

500
Client Node
Selected
Front
End
502
ICE-T Comm
Libraries
515
510
500
Selected
Server Back End
Server Back End
Front
End
(Java)
Server Node
520
530

FIG. 5

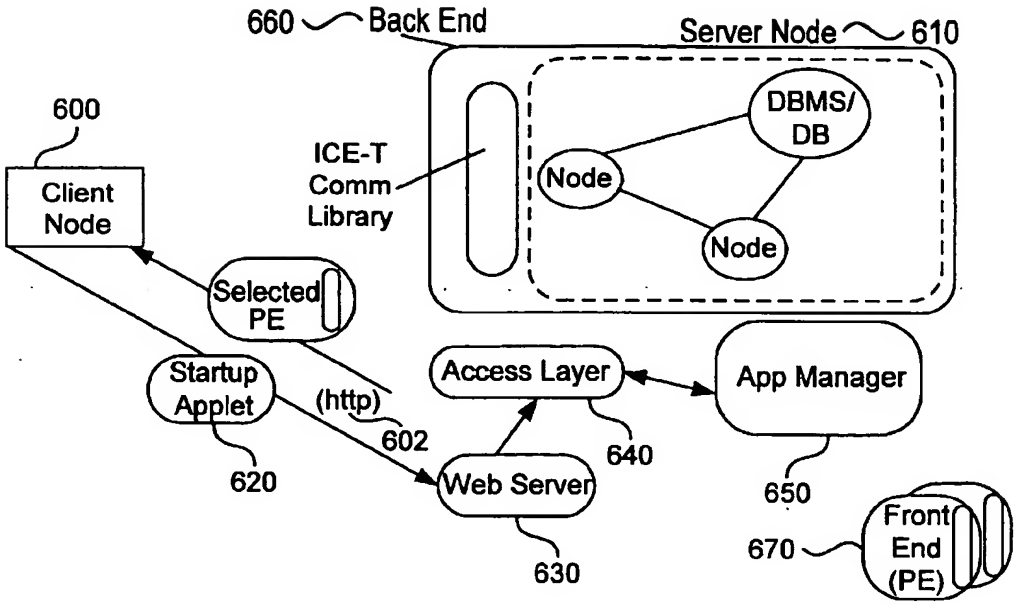

660
Back End
Server Node
610
600
Client Node
ICE-T Comm Library
DBMS/ DB
Node
Node
Selected PE
Startup Applet
(http)
602
620
Access Layer
640
App Manager
650
Web Server
630
Front End (PE)
670

FIG. 6

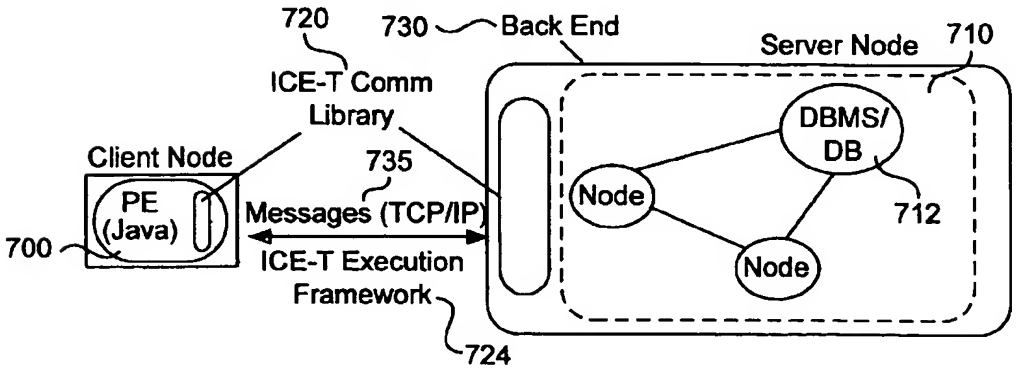
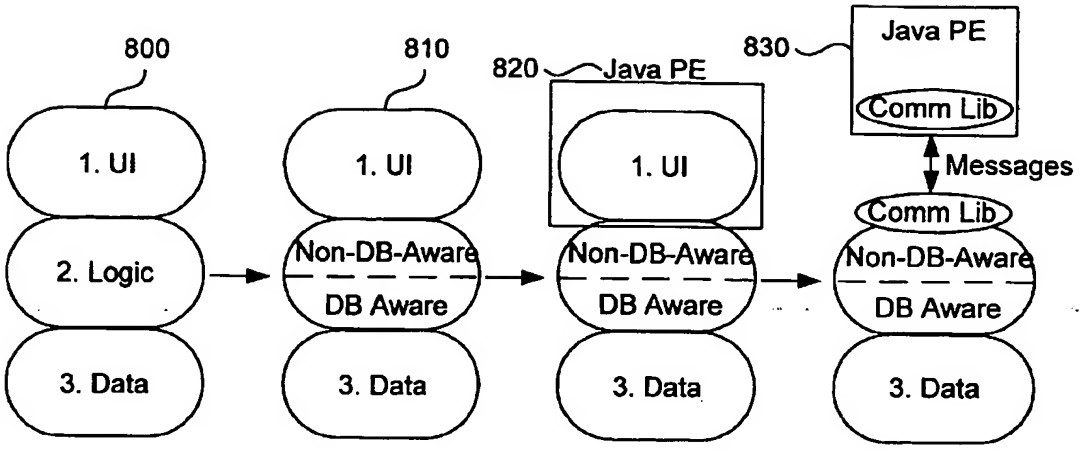

720
730
Back End
Server Node
710
ICE-T Comm Library
Client Node
735
PE (Java)
Messages (TCP/IP)
700
ICE-T Execution Framework
724
DBMS/ DB
712
Node
Node

FIG. 7



Step 1: Analyze Logic Step 2: Create PE Step 3: Define Messages

FIG. 8

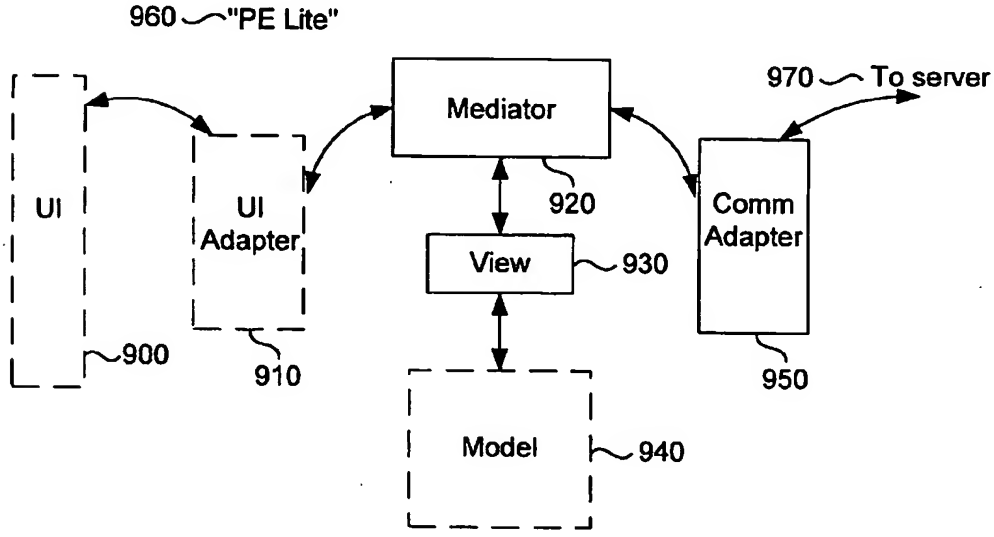


FIG. 9

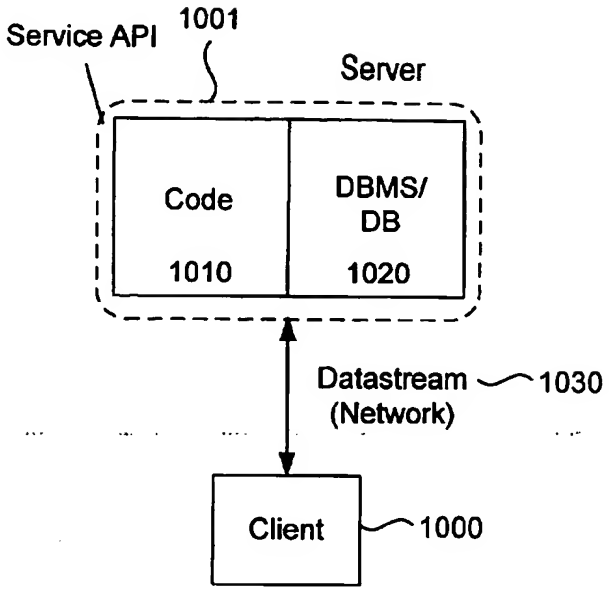


FIG. 10 (Prior Art)

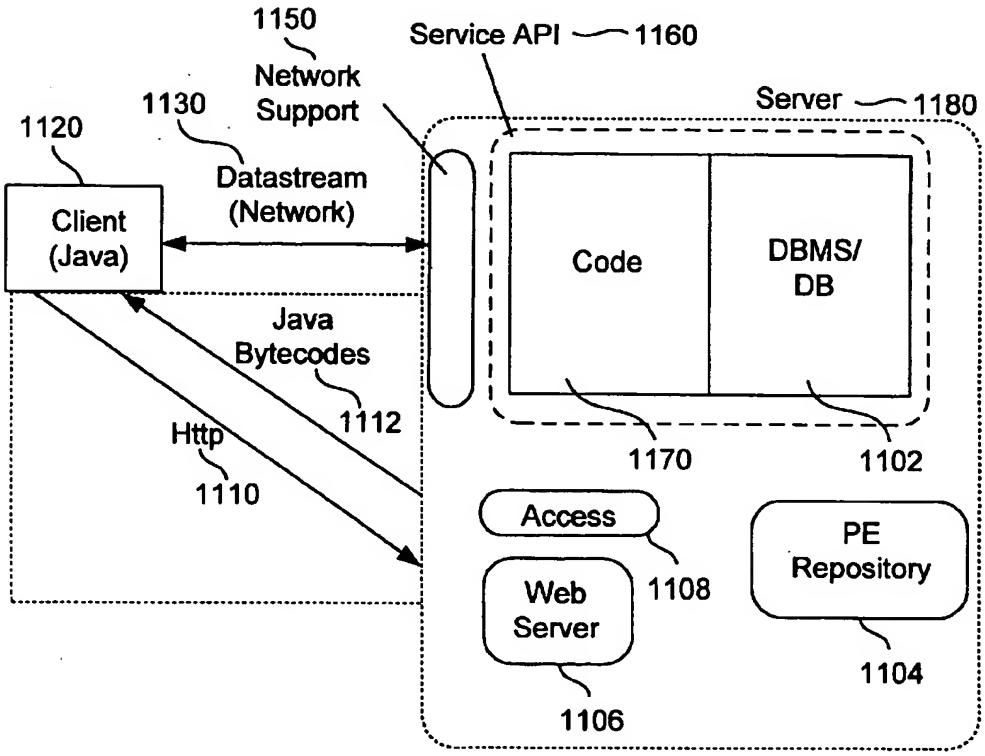


FIG. 11

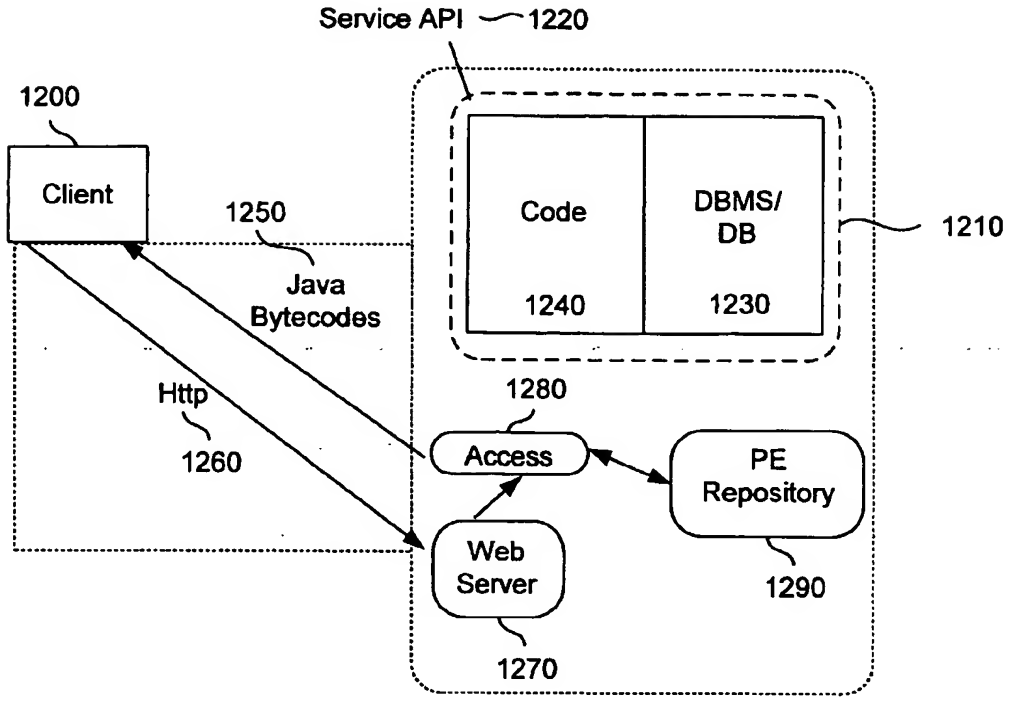

Service API
1220
1200
Client
1250
Java
Bytecodes
Code
DBMS/
DB
1210
1240
1230
Http
1260
1280
Access
PE
Repository
Web
Server
1290
1270

FIG. 12

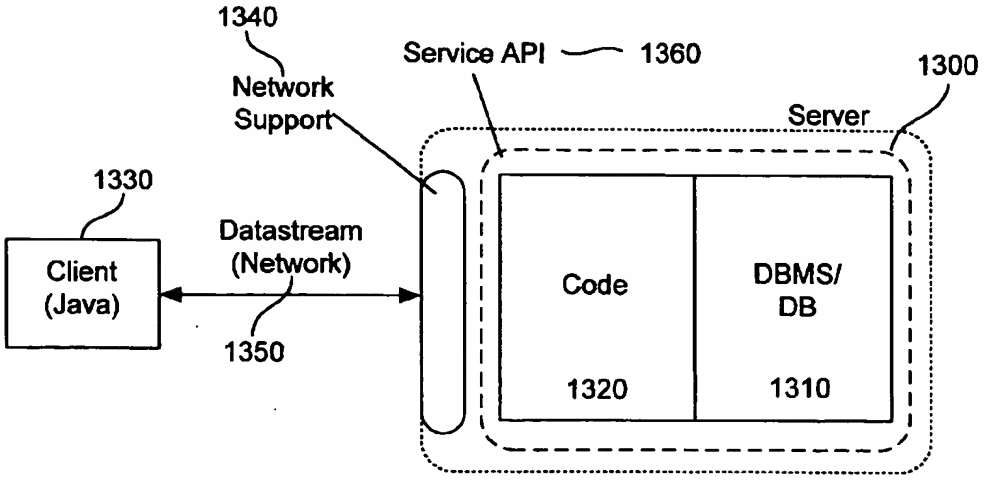

1340
Network
Support
Service API
1360
1300
Server
1330
Client
(Java)
Datastream
(Network)
1350
Code
DBMS/
DB
1320
1310

FIG. 13

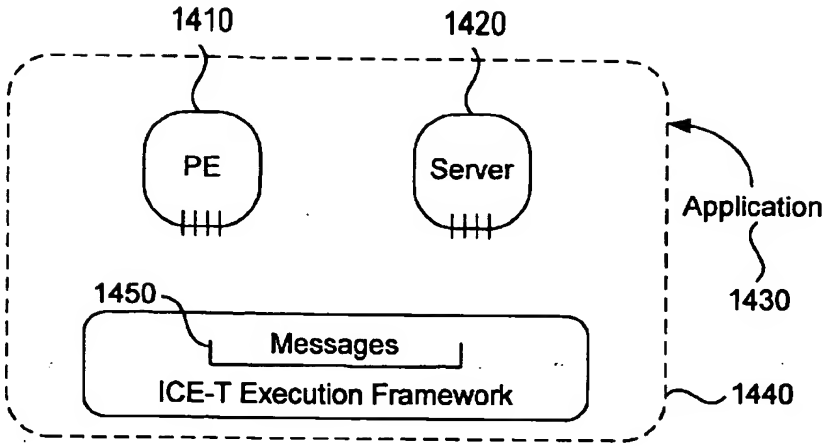

1410
1420
PE
Server
Application
1430
1450
Messages
ICE-T Execution Framework
1440

FIG. 14

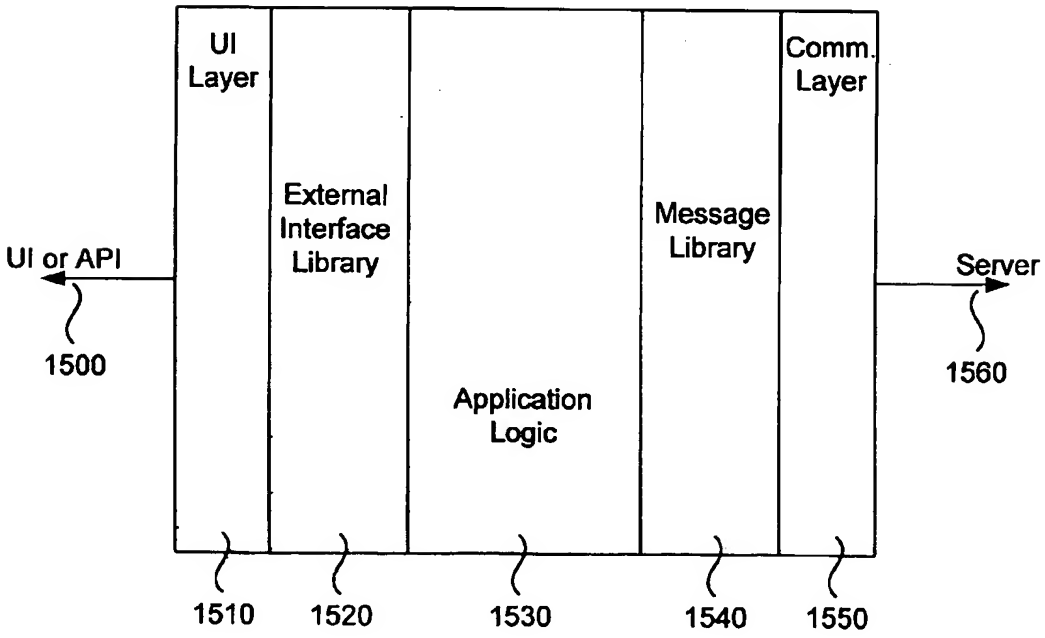

UI Layer
External Interface Library
Application Logic
Message Library
Comm. Layer
UI or API
1500
Server
1560
1510
1520
1530
1540
1550

FIG. 15

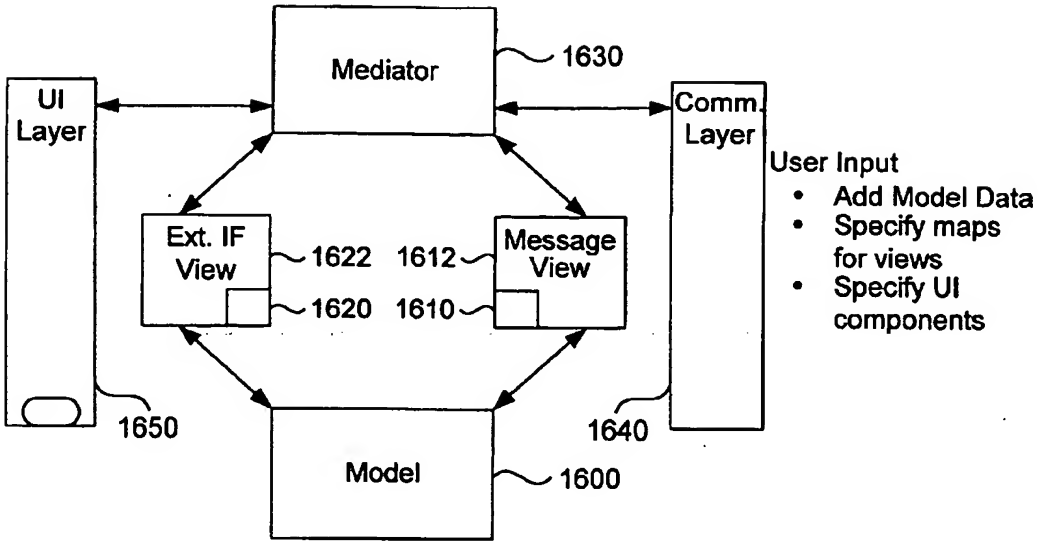

Mediator
1630
UI Layer
Comm. Layer
User Input
• Add Model Data
• Specify maps for views
• Specify UI components
Ext. IF View
1622
1612
Message View
1620
1610
1650
1640
Model
1600

FIG. 16

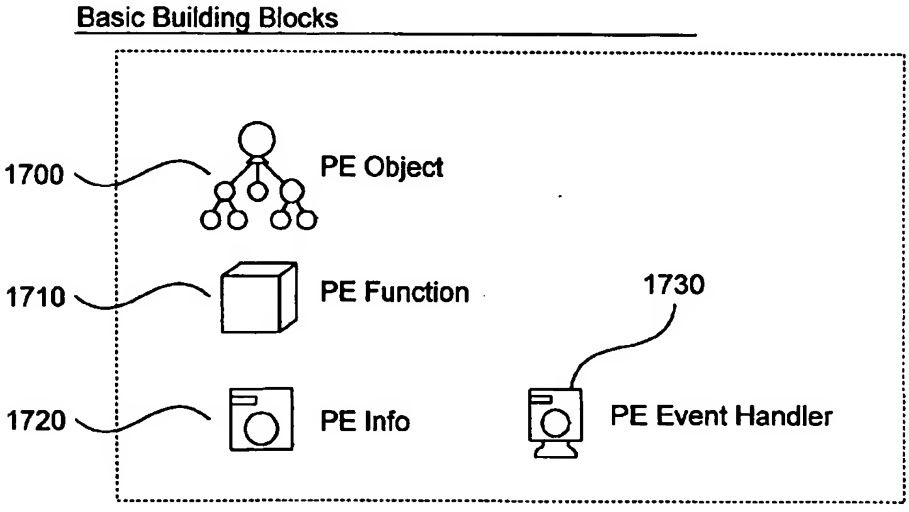

Basic Building Blocks
1700
PE Object
1710
PE Function
1730
1720
PE Info
PE Event Handler

FIG. 17

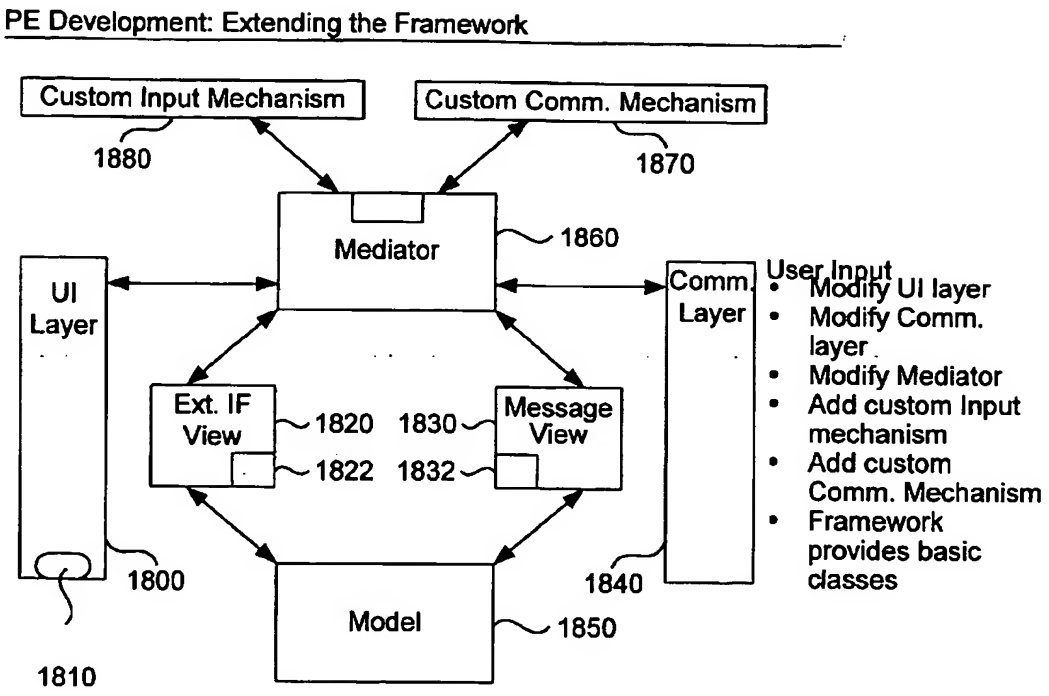

PE Development: Extending the Framework
Custom Input Mechanism
1880
Custom Comm. Mechanism
1870
Mediator
1860
UI Layer
Comm. Layer
User Input
• Modify UI layer
• Modify Comm. layer
• Modify Mediator
• Add custom Input mechanism
• Add custom Comm. Mechanism
• Framework provides basic classes
Ext. IF View
1820
1822
1830
1832
Message View
1800
1840
Model
1850
1810

FIG. 18

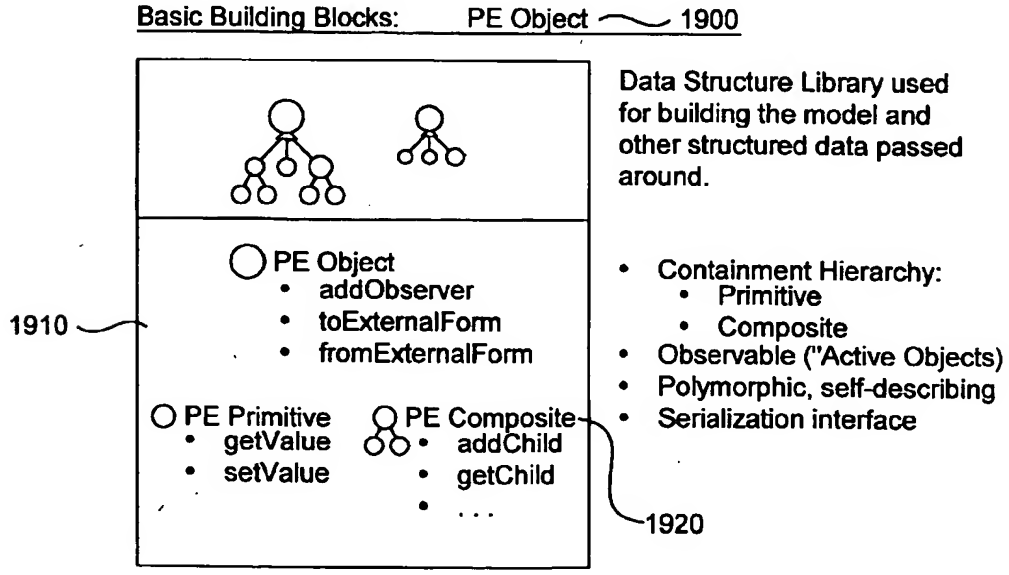

Basic Building Blocks: PE Object 1900
Data Structure Library used for building the model and other structured data passed around.
PE Object
• addObserver
• toExternalForm
• fromExternalForm
1910
PE Primitive
• getValue
• setValue
PE Composite
• addChild
• getChild
• ...
• Containment Hierarchy:
• Primitive
• Composite
• Observable ("Active Objects)
• Polymorphic, self-describing
• Serialization interface
1920

FIG. 19

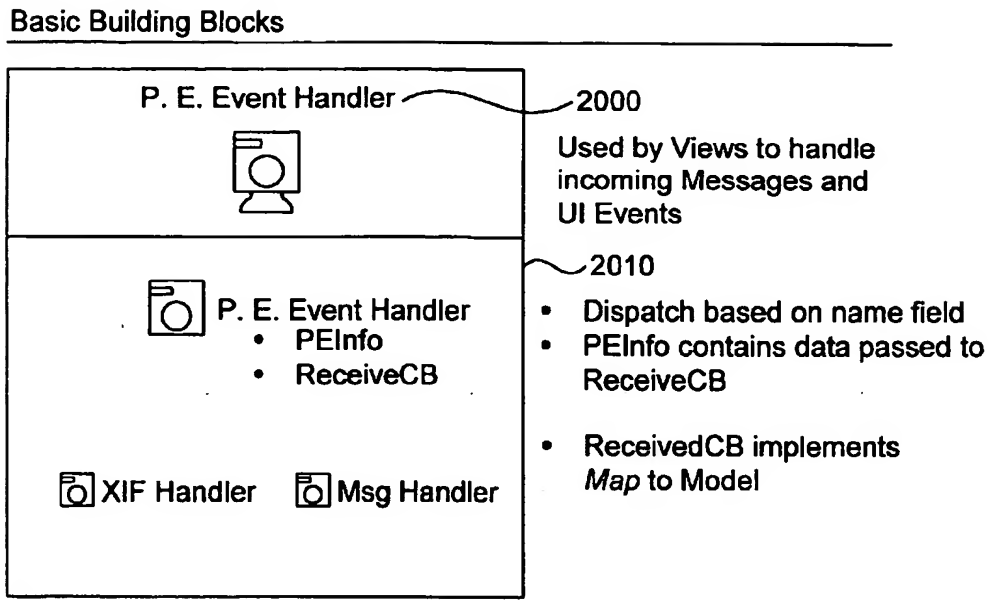

Basic Building Blocks
P. E. Event Handler
2000
Used by Views to handle incoming Messages and UI Events
2010
P. E. Event Handler
• PEInfo
• ReceiveCB
• Dispatch based on name field
• PEInfo contains data passed to ReceiveCB
• ReceivedCB implements Map to Model
XIF Handler
Msg Handler

FIG. 20

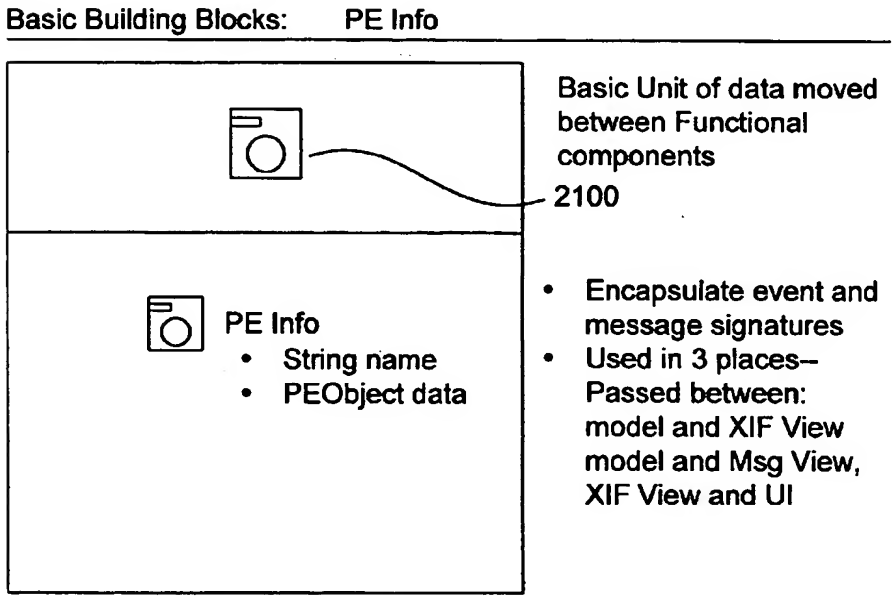

Basic Building Blocks: PE Info
Basic Unit of data moved between Functional components
2100
PE Info
• String name
• PEObject data
• Encapsulate event and message signatures
• Used in 3 places– Passed between: model and XIF View model and Msg View, XIF View and UI

FIG. 21

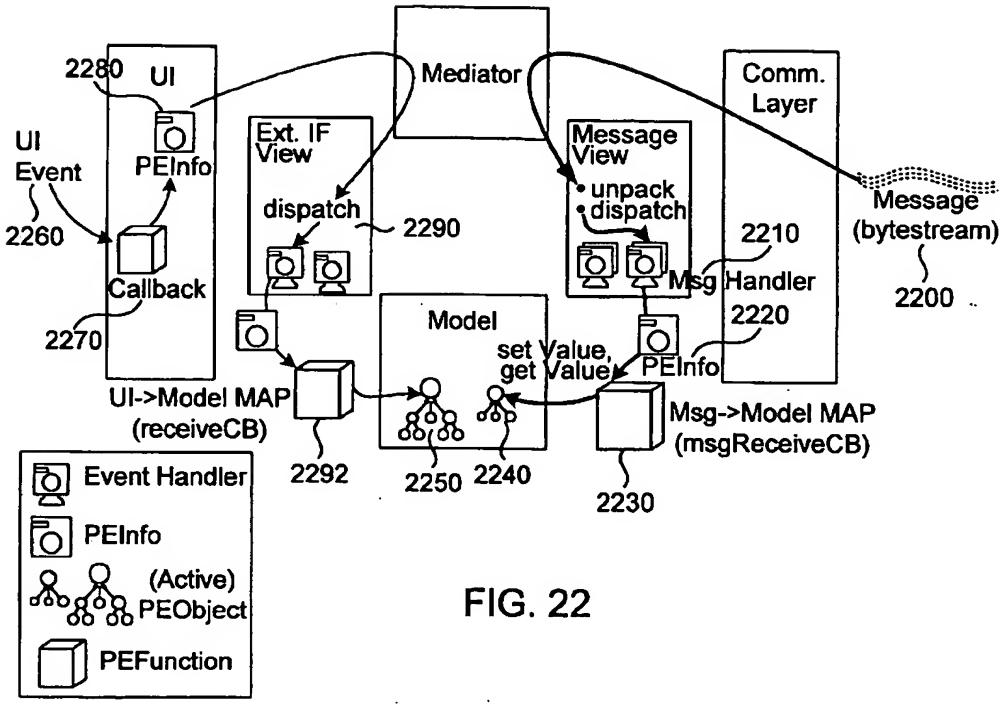

2280
UI
Mediator
Comm. Layer
UI Event
PEInfo
Ext. IF View
Message View
unpack
dispatch
Message (bytestream)
2260
dispatch
2290
2210
Msg Handler
2200
Callback
Model
2220
2270
set Value,
get Value
PEInfo
UI->Model MAP
(receiveCB)
Msg->Model MAP
(msgReceiveCB)
2292
2250
2240
2230
Event Handler
PEInfo
(Active)
PEObject
PEFunction

FIG. 22

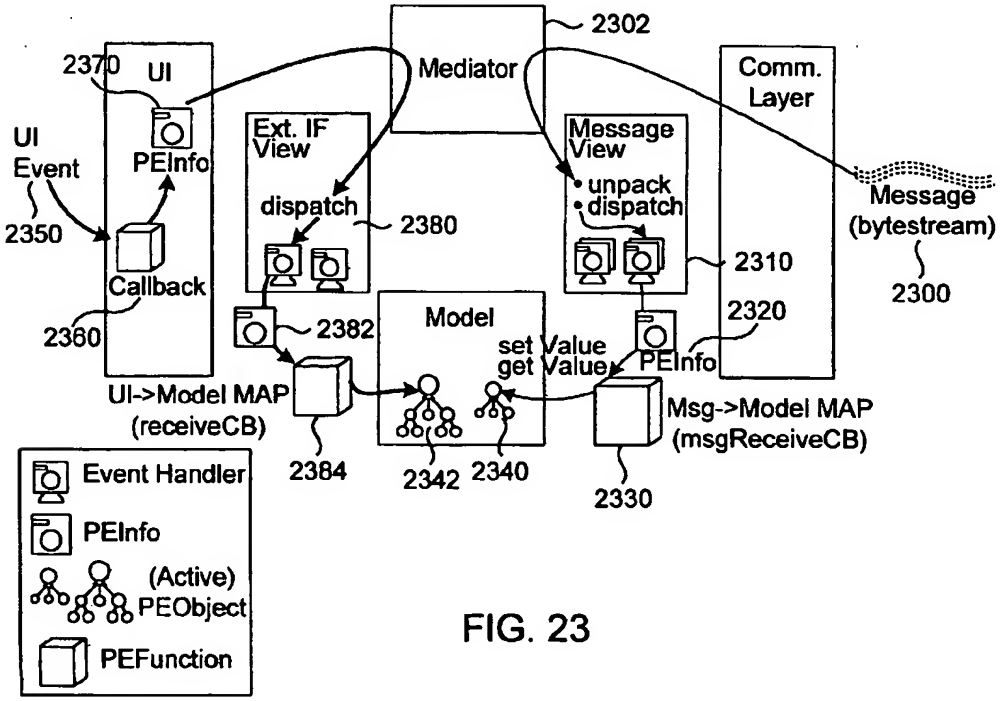

2302
2370
UI
Mediator
Comm. Layer
UI Event
PEInfo
Ext. IF View
Message View
unpack
dispatch
Message (bytestream)
2350
dispatch
2380
2310
2300
Callback
2382
Model
2320
2360
set Value
get Value
PEInfo
UI->Model MAP
(receiveCB)
Msg->Model MAP
(msgReceiveCB)
2384
2342
2340
2330
Event Handler
PEInfo
(Active)
PEObject
PEFunction

FIG. 23

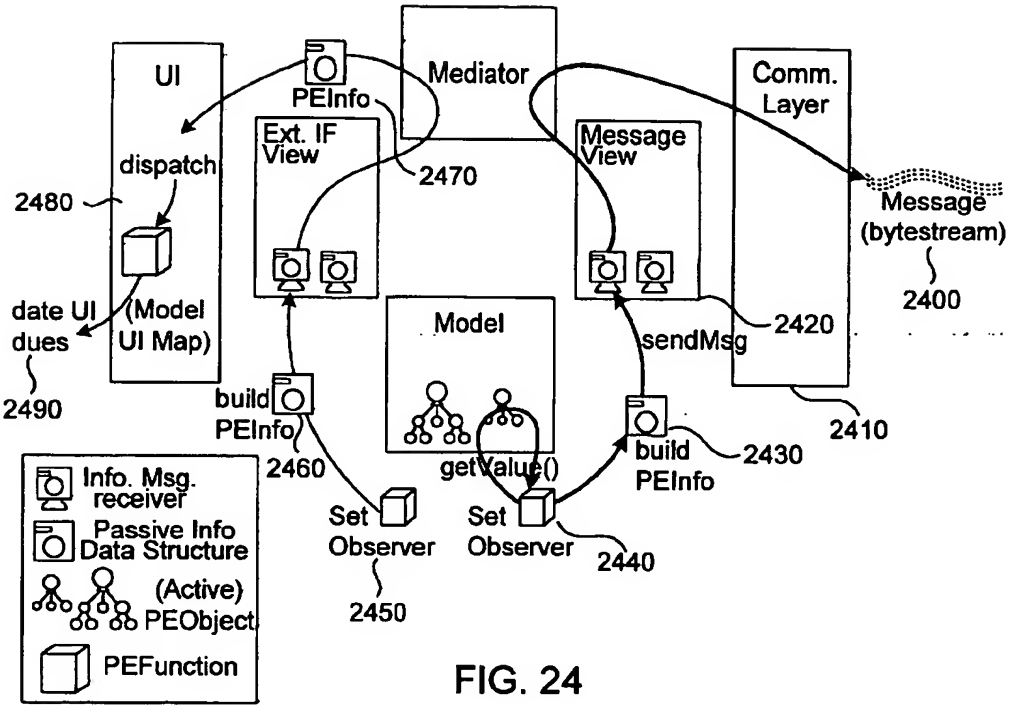

UI
dispatch
2480
date UI
dues
2490
(Model
UI Map)
PEInfo
Mediator
Ext. IF
View
2470
Message
View
Comm.
Layer
Message
(bytestream)
2400
Model
sendMsg
2420
build
PEInfo
2460
getValue()
build
PEInfo
2430
2410
Set
Observer
2450
Set
Observer
2440
Info. Msg.
receiver
Passive Info
Data Structure
(Active)
PEObject
PEFunction

FIG. 24

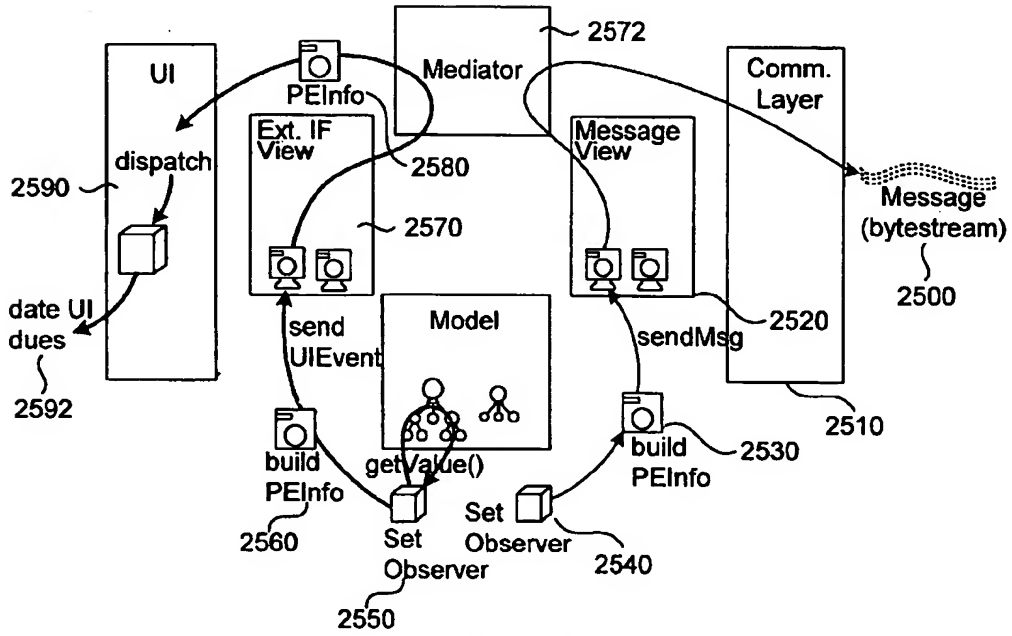

2572
UI
dispatch
2590
date UI
dues
2592
PEInfo
Mediator
Ext. IF
View
2580
2570
Message
View
Comm.
Layer
Message
(bytestream)
2500
send
UIEvent
Model
sendMsg
2520
build
PEInfo
2560
getValue()
Set
Observer
2550
Set
Observer
2540
build
PEInfo
2530
2510

FIG. 25

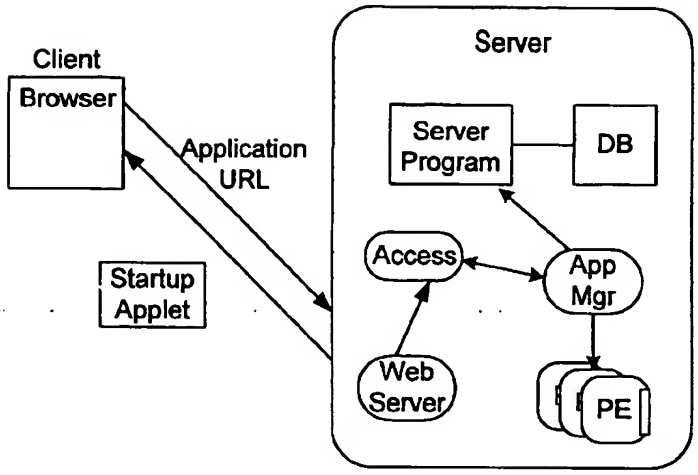


FIG. 26

Name	Description	Development	Runtime
PresentationEngine Class	Is the superclass for Presentation Engines.	Is derived from. Is not modified.	Is used by your instance of the Presentation Engine.
pe-template	Is an instance of the PresentationEngine class.	Is modified. Developers copy this file, give it a name, and write method bodies for a few methods.	
myPresentationEnqine (a developed and compiled Presentation Engine, given an example name here)	Is an instance of the PresentationEngine class, created using pe-template, and filled in with application-specific code.	Is developed from the pe-template.	A compiled Presentation Engine (here named myPresentation Enqine) that includes a GUI and an instance of the client Communication Library.

FIG. 27

Function	Description	Requirements
setSrvCommProperties()	Sets server Communication Library properties such as server timeouts, and debug file and trace level.	Should not write anything to stdout because the ICE-T Access Layer expects the first thing written to stdout to be the port number that the server uses to listen for the client connection.
createMessageHandlers()	Registers the message handlers. Is the server analog to the createMessageHandlers() method in the Presentation Engine.	Called immediately after the setSrvCommProperties () call.
initializeApplication()	Starts up the application-specific code. Is called after the server program accepts the connection from the client and before the calls for reading and handling messages.	Initializes the data structures required for handling messages from the client. Must return. If the server program has its own notifier loop, such as a loop that receives notifications from the database, then that notifier must be called in a new thread.

FIG. 28

Property	Default Accepted Values	
setServerTimeout()	3600 seconds (1 hour)	>0 Specifying 0 or <0 means no timeout
setDbgFile	s tderr	User-specified filename
setAcceptTxmeout	3600 seconds	>0 Specifying 0 or <0 means no hmeout
setTraceLevel	0 (zero)	0 means no trace messages are printed. 2 means trace messages are printed.
setThreadModel	SRVCOMM-THREAD-MODEL-SINGLE-HANDLER	SRVCOb~l- THREAD-MODEL-SINGLE HANDLER SRVCOI-THREAD-MODEL-MULTIPLE-HANDLERS

FIG. 29

Table D-1 describes client and server-side exceptions
Table D-1 ICE-T Exceptions

Exception	Description
Exceptions thrown by the client prol ram (Java code):	
PeException	A general exception that signifies an unidentitifed problem. The message may include more detail about where it was raised. Pe~.xcep~ion is used very rarely.
MapNotFoundException	Raised when a module in the Presentation Engine Receives a message for which no map function has been registered. The message fexplains where and why the exception was raised.
DuplicateMapException	Raised when the client program attempts to register the same message in both the UI and the Model.
CommException	A general communication exception. It is indicated that there has been some communication problem and prints a message with more details.
ConnectionException	Indicates that there were some problems in connecting to the server.
ICE-T protocol that	PeProtocolExceptionIndicates that there was a problem with the is being sent between the client and server.
Exceptions thrown by the Server Communication Library:	
ServerException	The base class of the other exceptions. Never thrown.
SrvCommException	A general communication exception. Indicates that there has been some communication problem and prints a message with more details.
SrvConnectionException	Indicates that there is a problem with the connection to the client. It usually indicates that either the client has shut down unexpectedly or the connection has terminated, and the server is somehow continuing on.
SrvProtocolException	Indicates that there has been a problem with the message sent to, or received from, the client. Either the message type is unknown, or there has been a problem marshalling or unmarshalling the data. It means that somehow the bytes that were sent on the socket were garbled.

FIG. 30

OBJECT-ORIENTED SYSTEM, METHOD AND ARTICLE OF MANUFACTURE FOR A CLIENT-SERVER SESSION WEB ACCESS IN AN INTERPRISE COMPUTING FRAMEWORK SYSTEM.

COPYRIGHT NOTIFICATION

Portions of this patent application contain materials that are subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document, or the patent disclosure, as it appears in the Patent and Trademark Office.

FIELD OF THE INVENTION

This invention generally relates to improvements in computer systems and, more particularly, to operating system software for managing Interprise computing in a network user interface.

BACKGROUND OF THE INVENTION

One of the most important aspects of a modern computing system is the interface between the human user and the machine. The earliest and most popular type of interface was text based; a user communicated with the machine by typing text characters on a keyboard and the machine communicated with the user by displaying text characters on a display screen. More recently, graphic user interfaces have become popular where the machine communicates with a user by displaying graphics, including text and pictures, on a display screen and the user communicates with the machine both by typing in textual commands and by manipulating the displayed pictures with a pointing device, such as a mouse.

Many modern computer systems operate with a graphic user interface called a window environment. In a typical window environment, the graphical display portrayed on the display screen is arranged to resemble the surface of an electronic "desktop" and each application program running on the computer is represented as one or more electronic "paper sheets" displayed in rectangular regions of the screen called "windows".

Each window region generally displays information which is generated by the associated application program and there may be several window regions simultaneously present on the desktop, each representing information generated by a different application program. An application program presents information to the user through each window by drawing or "painting" images, graphics or text within the window region. The user, in turn, communicates with the application by "pointing at" objects in the window region with a cursor which is controlled by a pointing device and manipulating the objects and also by typing information into the keyboard. The window regions may also be moved around on the display screen and changed in size and appearance so that the user can arrange the desktop in a convenient manner.

Each of the window regions also typically includes a number of standard graphical objects such as sizing boxes, buttons and scroll bars. These features represent user interface devices that the user can point at with the cursor to select and manipulate. When the devices are selected or manipulated, the underlying application program is informed, via the window system, that the control has been manipulated by the user.

In general, the window environment described above is part of the computer operating system. The operating system

also typically includes a collection of utility programs that enable the computer system to perform basic operations, such as storing and retrieving information on a disc memory, communicating with a network and performing file operations including the creation, naming and renaming of files and, in some cases, performing diagnostic operations in order to discover or recover from malfunctions.

The last part of the computing system is the "application program" which interacts with the operating system to provide much higher level functionality, perform a specific task and provide a direct interface with the user. The application program typically makes use of operating system functions by sending out series of task commands to the operating system which then performs a requested task. For example, the application program may request that the operating system store particular information on the computer disc memory or display information on the video display.

FIG. **1** is a schematic illustration of a typical prior art computer system utilizing both an application program and an operating system. The computer system is schematically represented by box **100**, the application is represented by box **102** and the operating system by box **106**. The previously-described interaction between the application program **102** and the operating system **106** is illustrated schematically by arrow **104**. This dual program system is used on many types of computer systems ranging from main frames to personal computers.

The method for handling screen displays varies from computer to computer and, in this regard, FIG. **1** represents a prior art personal computer system. In order to provide screen displays, application program **102** generally stores information to be displayed (the storing operation is shown schematically by arrow **108**) into a screen buffer **110**. Under control of various hardware and software in the system the contents of the screen buffer **110** are read out of the buffer and provided, as indicated schematically by arrow **114**, to a display adapter **112**. The display adapter **112** contains hardware and software (sometimes in the form of firmware) which converts the information in screen buffer **110** to a form which can be used to drive the display monitor **118** which is connected to display adapter **112** by cable **116**.

The prior art configuration shown in FIG. **1** generally works well in a system where a single application program **102** is running at any given time. This simple system works properly because the single application program **102** can write information into any area of the entire screen buffer area **110** without causing a display problem. However, if the configuration shown in FIG. **1** is used in a computer system where more than one application program **102** can be operational at the same time (for example, a "multi-tasking" computer system) display problems can arise. More particularly, if each application program has access to the entire screen buffer **110**, in the absence of some direct communication between applications, one application may overwrite a portion of the screen buffer which is being used by another application, thereby causing the display generated by one application to be overwritten by the display generated by the other application.

Accordingly, mechanisms were developed to coordinate the operation of the application programs to ensure that each application program was confined to only a portion of the screen buffer thereby separating the other displays. This coordination became complicated in systems where windows were allowed to "overlap" on the screen display. When the screen display is arranged so that windows appear to

"overlap", a window which appears on the screen in "front" of another window covers and obscures part of the underlying window. Thus, except for the foremost window, only part of the underlying windows may be drawn on the screen and be "visible" at any given time. Further, because the windows can be moved or resized by the user, the portion of each window which is visible changes as other windows are moved or resized. Thus, the portion of the screen buffer which is assigned to each application window also changes as windows from other applications are moved or resized.

In order to efficiently manage the changes to the screen buffer necessary to accommodate rapid screen changes caused by moving or resizing windows, the prior art computer arrangement shown in FIG. **1** was modified as shown in FIG. **2**. In this new arrangement computer system **200** is controlled by one or more application programs, of which programs **202** and **216** are shown, which programs may be running simultaneously in the computer system. Each of the programs interfaces with the operating system **204** as illustrated schematically by arrows **206** and **220**. However, in order to display information on the display screen, application programs **202** and **216** send display information to a central window manager program **218** located in the operating system **204**. The window manager program **218**, in turn, interfaces directly with the screen buffer **210** as illustrated schematically by arrow **208**. The contents of screen buffer **210** are provided, as indicated by arrow **212**, to a display adapter **214** which is connected by a cable **222** to a display monitor **224**.

In such a system, the window manager **218** is generally responsible for maintaining all of the window displays that the user views during operation of the application programs. Since the window manager **218** is in communication with all application programs, it can coordinate between applications to insure that window displays do not overlap. Consequently, it is generally the task of the window manager to keep track of the location and size of the window and the window areas which must be drawn and redrawn as windows are moved.

The window manager **218** receives display requests from each of the applications **202** and **216**. However, since only the window manager **218** interfaces with the screen buffer **210**, it can allocate respective areas of the screen buffer **210** for each application and insure that no application erroneously overwrites the display generated by another application. There are a number of different window environments commercially available which utilize the arrangement illustrated in FIG. **2**. These include the X/Window Operating environment, the WINDOWS, graphical user interface developed by the Microsoft Corporation and the OS/2 Presentation Manager, developed by the International Business Machines Corporation, and the Macintosh OS, developed by Apple Computer Corporation.

Each of these window environments has its own internal software architecture, but the architectures can all be classified by using a multi-layer model similar to the multi-layer models used to describe computer network software. A typical multi-layer model includes the following layers:

User Interface
Window Manager
Resource Control and Communication
Component Driver Software
Computer Hardware

where the term "window environment" refers to all of the above layers taken together.

The lowest or computer hardware level includes the basic computer and associated input and output devices including display monitors, keyboards, pointing devices, such as mice or trackballs, and other standard components, including printers and disc drives. The next or "component driver software" level consists of device-dependent software that generates the commands and signals necessary to operate the various hardware components. The resource control and communication layer interfaces with the component drivers and includes software routines which allocate resources, communicate between applications and multiplex communications generated by the higher layers to the underlying layers. The window manager handles the user interface to basic window operations, such as moving and resizing windows, activating or inactivating windows and redrawing and repainting windows. The final user interface layer provides high level facilities that implement the various controls (buttons, sliders, boxes and other controls) that application programs use to develop a complete user interface.

Although the arrangement shown in FIG. **2** solves the display screen interference problem, it suffers from the drawback that the window manager **218** must process the screen display requests generated by all of the application programs. Since the requests can only be processed serially, the requests are queued for presentation to the window manager before each request is processed to generate a display on terminal **224**. In a display where many windows are present simultaneously on the screen, the window manager **218** can easily become a "bottleneck" for display information and prevent rapid changes of the display by the application programs **202** and **216**. A delay in the redrawing of the screen when windows are moved or repositioned by the user often manifests itself by the appearance that the windows are being constructed in a piecemeal fashion which becomes annoying and detracts from the operation of the system.

This problem becomes even more accentuated in a client-server environment where many applications are all in contention for very limited resources. The Internet has permeated the workplace as a communication medium of choice. Since the internet is accessible from almost any point in a typical business enterprise a new buzzword has evolved from the form enterprise computer into an "enterprise" computer. Interprise is a concatenation of internet and enterprise.

In today's client server enterprises, applications that exist in current client server enterprises are not really built to be managed since they are architected for distributed system environments. New systems are also required to be evolutionary, not revolutionary. Redesign of current systems that require significant expense need to be avoided.

A system is required that allows a user to create manageable applications, that can be readily deployed, installed on a variety of platforms, and configured to facilitate partitioning them on clients versus servers and administer the applications once they're running. Systems don't always break because of failure, errors, or bugs, they sometimes break because the enterprise itself is complicated and somebody does something unexpected somewhere which will bring the whole system down. When the system does come down, then a system administrator must be able to readily identify the problems, and deal with them in an effective manner so that a business doesn't stop functioning when one of these unforeseen events happens.

The application should be designed based on domain requirements, so it is independent of any platform underneath, and fits more with how commercial developers work. In the commercial world, the development process

isn't that important. The regular employees are not applications developers or programmers. Companies usually hire such work out; they get consultants to do that kind of work. Depending on the company and what they want done, it usually hires a consulting firm, individual consultants, or smaller groups of consultants to come in, help it develop an application. Their goal is the end application, which must be maintained. The company configures it, evolves it, and grows it. To allow for modification, the development task must be modular to allow different groups of people working on different parts of an application, without requiring any one group to understand every detail of the whole application of the enterprise.

The second criterion requires minimal extra knowledge, burden or sophistication on the part of the people developing the system. Most companies do not desire to have their business hinge on a single individual. Rather, they desire to have people who are primarily domain experts who can work with well-understood tools to produce a application matching company requirements quickly without making special demands on the company.

SUMMARY OF THE INVENTION

The foregoing problems are overcome in an illustrative embodiment of the invention in which an application is composed of a client (front end) program which communicates utilizing a network with a server (back end) program. The client and server programs are loosely coupled and exchange information using the network. The client program is composed of a User Interface (UI) and an object-oriented framework (Presentation Engine (PE) framework). The UI exchanges data messages with the framework. The framework is designed to handle two types of messages: (1) from the UI, and (2) from the server (back end) program via the network. The framework includes a component, the mediator which manages messages coming into and going out of the framework. A distributed computer system is disclosed with software for a client computer, a server computer and a network for connecting the client computer to the server computer which utilize an execution framework code segment configured to couple the server computer and the client computer via the network, by a plurality of client computer code segments resident on the server, each for transmission over the network to a client computer to initiate coupling; and a plurality of server computer code segments resident on the server which execute on the server in response to initiation of coupling via the network with a particular client utilizing the transmitted client computer code segment for communicating via a particular communication protocol. Communication is initiated utilizing the network to acquire characteristics of the client from the network.

BRIEF DESCRIPTION OF THE DRAWINGS

The above and further advantages of the invention may be better understood by referring to the following description in conjunction with the accompanying drawings, in which:

FIG. **1** is a schematic block diagram of a prior art computer system showing the relationship of the application program, the operating system, the screen buffer and the display monitor;

FIG. **2** is a schematic block diagram of a modification of the prior art system shown in FIG. **1** which allows several application programs running simultaneously to generate screen displays;

FIG. **3** is a schematic block diagram of a typical hardware configuration of a computer in accordance with the subject invention;

FIG. **4** is a block diagram in accordance with a preferred embodiment in an Enterprise Network;

FIG. **5** illustrates how a preferred embodiment leverages Java to facilitate the establishment and implementation of sever-centric policies;

FIG. **6** illustrates the processing associated with application startup in accordance with a preferred embodiment;

FIG. **7** illustrates the three fundamental components of an application in accordance with a preferred embodiment;

FIG. **8** illustrates the migration of an existing client-server application to one supported by a preferred embodiment;

FIG. **9** is a block diagram illustrating a Presentation Engine in accordance with a preferred embodiment;

FIG. **10** is a block diagram of a prior art client server architecture;

FIG. **11** illustrates an application in accordance with an alternate embodiment;

FIG. **12** illustrates a server establishing contact with a client in accordance with an alternate embodiment;

FIG. **13** illustrates a loosely coupled client-server application in accordance with an alternate embodiment;

FIG. **14** illustrates the system integration task necessary to develop an application in accordance with a preferred embodiment;

FIG. **15** is a block diagram illustrating the modular design of a client application in accordance with a preferred embodiment;

FIG. **16** is a block diagram of a framework in accordance with an alternate embodiment;

FIG. **17** illustrates the basic building blocks in accordance with an alternate embodiment;

FIG. **18** is a block diagram highlighting the steps utilized to extend the framework in accordance with a preferred embodiment;

FIG. **19** is an illustration of a Presentation Engine Object in accordance with a preferred embodiment;

FIG. **20** is an illustration of a Presentation Engine Event Handler used by Views to handle incoming messages and User Interface events in accordance with a preferred embodiment;

FIG. **21** illustrates a PEInfo object data in accordance with a preferred embodiment;

FIG. **22** illustrates incoming message flow to a model in accordance with a preferred embodiment;

FIG. **23** illustrates incoming messages mapping a User Interface to a Model in accordance with a preferred embodiment;

FIG. **24** illustrates outgoing messages mapping a model to messages in accordance with a preferred embodiment;

FIG. **25** illustrates outgoing messages mapping a model to a User Interface in accordance with a preferred embodiment;

FIG. **26** illustrates an arrangement that facilitates the launch and utilization of an application URL in accordance with a preferred embodiment;

FIG. **27** is a table that describes the forms of a Presentation Engine, as an abstract Java class, a template for development, and an executable component in an application in accordance with a preferred embodiment;

FIG. **28** describes the functions developers must fill in using the server program template in accordance with a preferred embodiment;

FIG. **29** illustrates Server Properties in accordance with a preferred embodiment; and

FIG. 30 is a table of client and server side exceptions in accordance with a preferred embodiment.

DETAILED DESCRIPTION

The invention is preferably practiced in the context of an operating system resident on a computer such as a SUN, IBM, PS/2, or Apple, Macintosh, computer. A representative hardware environment is depicted in FIG. 3, which illustrates a typical hardware configuration of a computer 300 in accordance with the subject invention. The computer 300 is controlled by a central processing unit 302 (which may be a conventional microprocessor) and a number of other units, all interconnected via a system bus 308, are provided to accomplish specific tasks. Although a particular computer may only have some of the units illustrated in FIG. 3, or may have additional components not shown, most computers will include at least the units shown.

Specifically, computer 300 shown in FIG. 3 includes a random access memory (RAM) 306 for temporary storage of information, a read only memory (ROM) 304 for permanent storage of the computer's configuration and basic operating commands and an input/output (I/O) adapter 310 for connecting peripheral devices such as a disk unit 313 and printer 314 to the bus 308, via cables 315 and 312, respectively. A user interface adapter 316 is also provided for connecting input devices, such as a keyboard 320, and other known interface devices including mice, speakers and microphones to the bus 308. Visual output is provided by a display adapter 318 which connects the bus 308 to a display device 322, such as a video monitor. The computer has resident thereon and is controlled and coordinated by operating system software such as the SUN Solaris or JavaOS operating system.

In a preferred embodiment, the invention is implemented in the C++ programming language using object-oriented programming techniques. C++ is a compiled language, that is, programs are written in a human-readable script and this script is then provided to another program called a compiler which generates a machine-readable numeric code that can be loaded into, and directly executed by, a computer. As described below, the C++ language has certain characteristics which allow a software developer to easily use programs written by others while still providing a great deal of control over the reuse of programs to prevent their destruction or improper use. The C++ language is well-known and many articles and texts are available which describe the language in detail. In addition, C++ compilers are commercially available from several vendors including Borland International, Inc. and Microsoft Corporation. Accordingly, for reasons of clarity, the details of the C++ language and the operation of the C++ compiler will not be discussed further in detail herein.

As will be understood by those skilled in the art, Object-Oriented Programming (OOP) techniques involve the definition, creation, use and destruction of "objects". These objects are software entities comprising data elements and routines, or functions, which manipulate the data elements. The data and related functions are treated by the software as an entity and can be created, used and deleted as if they were a single item. Together, the data and functions enable objects to model virtually any real-world entity in terms of its characteristics, which can be represented by the data elements, and its behavior, which can be represented by its data manipulation functions. In this way, objects can model concrete things like people and computers, and they can also model abstract concepts like numbers or geometrical designs.

Objects are defined by creating "classes" which are not objects themselves, but which act as templates that instruct the compiler how to construct the actual object. A class may, for example, specify the number and type of data variables and the steps involved in the functions which manipulate the data. An object is actually created in the program by means of a special function called a constructor which uses the corresponding class definition and additional information, such as arguments provided during object creation, to construct the object. Likewise objects are destroyed by a special function called a destructor. Objects may be used by using their data and invoking their functions.

The principle benefits of object-oriented programming techniques arise out of three basic principles; encapsulation, polymorphism and inheritance. More specifically, objects can be designed to hide, or encapsulate, all, or a portion of, the internal data structure and the internal functions. More particularly, during program design, a program developer can define objects in which all or some of the data variables and all or some of the related functions are considered "private" or for use only by the object itself. Other data or functions can be declared "public" or available for use by other programs. Access to the private variables by other programs can be controlled by defining public functions for an object which access the object's private data. The public functions form a controlled and consistent interface between the private data and the "outside" world. Any attempt to write program code which directly accesses the private variables causes the compiler to generate an error, which stops the compilation process and prevents the program from being run.

Polymorphism is a concept which allows objects and functions which have the same overall format, but which work with different data, to function differently in order to produce consistent results. For example, an addition function may be defined as variable A plus variable B (A+B) and this same format can be used whether the A and B are numbers, characters or dollars and cents. However, the actual program code which performs the addition may differ widely depending on the type of variables that comprise A and B. Polymorphism allows three separate function definitions to be written, one for each type of variable (numbers, characters and dollars). After the functions have been defined, a program can later refer to the addition function by its common format (A+B) and, during compilation, the C++ compiler will determine which of the three functions is actually being used by examining the variable types. The compiler will then substitute the proper function code. Polymorphism allows similar functions which produce analogous results to be "grouped" in the program source code to produce a more logical and clear program flow.

The third principle which underlies object-oriented programming is inheritance, which allows program developers to easily reuse pre-existing programs and to avoid creating software from scratch. The principle of inheritance allows a software developer to declare classes (and the objects which are later created from them) as related. Specifically, classes may be designated as subclasses of other base classes. A subclass "inherits" and has access to all of the public functions of its base classes just as if these function appeared in the subclass. Alternatively, a subclass can override some or all of its inherited functions or may modify some or all of its inherited functions merely by defining a new function with the same form (overriding or modification does not alter the function in the base class, but merely modifies the use of the function in the subclass). The creation of a new subclass which has some of the functionality (with selective

modification) of another class allows software developers to easily customize existing code to meet their particular needs.

Although object-oriented programming offers significant improvements over other programming concepts, program development still requires significant outlays of time and effort, especially if no pre-existing software programs are available for modification. Consequently, a prior art approach has been to provide a program developer with a set of pre-defined, interconnected classes which create a set of objects and additional miscellaneous routines that are all directed to performing commonly-encountered tasks in a particular environment. Such pre-defined classes and libraries are typically called "frameworks" and essentially provide a pre-fabricated structure for a working application.

For example, a framework for a user interface might provide a set of pre-defined graphic interface objects which create windows, scroll bars, menus, etc. and provide the support and "default" behavior for these graphic interface objects. Since frameworks are based on object-oriented techniques, the pre-defined classes can be used as base classes and the built-in default behavior can be inherited by developer-defined subclasses and either modified or overridden to allow developers to extend the framework and create customized solutions in a particular area of expertise. This object-oriented approach provides a major advantage over traditional programming since the programmer is not changing the original program, but rather extending the capabilities of the original program. In addition, developers are not blindly working through layers of code because the framework provides architectural guidance and modeling and, at the same time, frees the developers to supply specific actions unique to the problem domain.

There are many kinds of frameworks available, depending on the level of the system involved and the kind of problem to be solved. The types of frameworks range from high-level application frameworks that assist in developing a user interface, to lower-level frameworks that provide basic system software services such as communications, printing, file systems support, graphics, etc. Commercial examples of application frameworks include MacApp (Apple), Bedrock (Symantec), OWL (Borland), NeXT Step App Kit (NeXT), and Smalltalk-80 MVC (ParcPlace).

While the framework approach utilizes all the principles of encapsulation, polymorphism, and inheritance in the object layer, and is a substantial improvement over other programming techniques, there are difficulties which arise. Application frameworks generally consist of one or more object "layers" on top of a monolithic operating system and even with the flexibility of the object layer, it is still often necessary to directly interact with the underlying operating system by means of awkward procedural calls.

In the same way that an application framework provides the developer with prefabricated functionality for an application program, a system framework, such as that included in a preferred embodiment, can provide a prefabricated functionality for system level services which developers can modify or override to create customized solutions, thereby avoiding the awkward procedural calls necessary with the prior art application frameworks programs. For example, consider a display framework which could provide the foundation for creating, deleting and manipulating windows to display information generated by an application program. An application software developer who needed these capabilities would ordinarily have to write specific routines to provide them. To do this with a framework, the developer only needs to supply the characteristics and behavior of the finished display, while the framework provides the actual routines which perform the tasks.

A preferred embodiment takes the concept of frameworks and applies it throughout the entire system, including the application and the operating system. For the commercial or corporate developer, systems integrator, or OEM, this means all of the advantages that have been illustrated for a framework such as MacApp can be leveraged not only at the application level for such things as text and user interfaces, but also at the system level, for services such as printing, graphics, multi-media, file systems, I/O, testing, etc.

A preferred embodiment is written using JAVA, C, and the C++ language and utilizes object-oriented programming methodology. Object-oriented programming (OOP) has become increasingly used to develop complex applications. As OOP moves toward the mainstream of software design and development, various software solutions require adaptation to make use of the benefits of OOP. A need exists for these OOP principles to be applied to a messaging interface of an electronic messaging system such that a set of OOP classes and objects for the messaging interface can be provided.

OOP is a process of developing computer software using objects, including the steps of analyzing the problem, designing the system, and constructing the program. An object is a software package that contains both data and a collection of related structures and procedures. Since it contains both data and a collection of structures and procedures, it can be visualized as a self-sufficient component that does not require other additional structures, procedures or data to perform its specific task. OOP, therefore, views a computer program as a collection of largely autonomous components, called objects, each of which is responsible for a specific task. This concept of packaging data, structures, and procedures together in one component or module is called encapsulation.

In general, OOP components are reusable software modules which present an interface that conforms to an object model and which are accessed at run-time through a component integration architecture. A component integration architecture is a set of architecture mechanisms which allow software modules in different process spaces to utilize each others' capabilities or functions. This is generally done by assuming a common component object model on which to build the architecture.

It is worthwhile to differentiate between an object and a class of objects at this point. An object is a single instance of the class of objects, which is often just called a class. A class of objects can be viewed as a blueprint, from which many objects can be formed.

OOP allows the programmer to create an object that is a part of another object. For example, the object representing a piston engine is said to have a composition-relationship with the object representing a piston. In reality, a piston engine comprises a piston, valves and many other components; the fact that a piston is an element of a piston engine can be logically and semantically represented in OOP by two objects.

OOP also allows creation of an object that "depends from" another object. If there are two objects, one representing a piston engine and the other representing a piston engine wherein the piston is made of ceramic, then the relationship between the two objects is not that of composition. A ceramic piston engine does not make up a piston engine. Rather it is merely one kind of piston engine that has one more limitation than the piston engine; its piston is made

of ceramic. In this case, the object representing the ceramic piston engine is called a derived object, and it inherits all of the aspects of the object representing the piston engine and adds further limitation or detail to it. The object representing the ceramic piston engine "depends from" the object representing the piston engine. The relationship between these objects is called inheritance.

When the object or class representing the ceramic piston engine inherits all of the aspects of the objects representing the piston engine, it inherits the thermal characteristics of a standard piston defined in the piston engine class. However, the ceramic piston engine object overrides these ceramic specific thermal characteristics, which are typically different from those associated with a metal piston. It skips over the original and uses new functions related to ceramic pistons. Different kinds of piston engines have different characteristics, but may have the same underlying functions associated with it (e.g., how many pistons in the engine, ignition sequences, lubrication, etc.). To access each of these functions in any piston engine object, a programmer would call the same functions with the same names, but each type of piston engine may have different/overriding implementations of functions behind the same name. This ability to hide different implementations of a function behind the same name is called polymorphism and it greatly simplifies communication among objects.

With the concepts of composition-relationship, encapsulation, inheritance and polymorphism, an object can represent just about anything in the real world. In fact, our logical perception of the reality is the only limit on determining the kinds of things that can become objects in object-oriented software. Some typical categories are as follows:

- Objects can represent physical objects, such as automobiles in a traffic-flow simulation, electrical components in a circuit-design program, countries in an economics model, or aircraft in an air-traffic-control system.
- Objects can represent elements of the computer-user environment such as windows, menus or graphics objects.
- An object can represent an inventory, such as a personnel file or a table of the latitudes and longitudes of cities.
- An object can represent user-defined data types such as time, angles, and complex numbers, or points on the plane.

With this enormous capability of an object to represent just about any logically separable matters, OOP allows the software developer to design and implement a computer program that is a model of some aspects of reality, whether that reality is a physical entity, a process, a system, or a composition of matter. Since the object can represent anything, the software developer can create an object which can be used as a component in a larger software project in the future.

If 90% of a new OOP software program consists of proven, existing components made from preexisting reusable objects, then only the remaining 10% of the new software project has to be written and tested from scratch. Since 90% already came from an inventory of extensively tested reusable objects, the potential domain from which an error could originate is 10% of the program. As a result, OOP enables software developers to build objects out of other, previously built, objects.

This process closely resembles complex machinery being built out of assemblies and sub-assemblies. OOP technology, therefore, makes software engineering more like hardware engineering in that software is built from existing components, which are available to the developer as objects. All this adds up to an improved quality of the software as well as an increased speed of its development.

Programming languages are beginning to fully support the OOP principles, such as encapsulation, inheritance, polymorphism, and composition-relationship. With the advent of the C++ language, many commercial software developers have embraced OOP. C++ is an OOP language that offers a fast, machine-executable code. Furthermore, C++ is suitable for both commercial-application and systems-programming projects. For now, C++ appears to be the most popular choice among many OOP programmers, but there is a host of other OOP languages, such as Smalltalk, common lisp object system (CLOS), and Eiffel. Additionally, OOP capabilities are being added to more traditional popular computer programming languages such as Pascal.

The benefits of object classes can be summarized, as follows:

- Objects and their corresponding classes break down complex programming problems into many smaller, simpler problems.
- Encapsulation enforces data abstraction through the organization of data into small, independent objects that can communicate with each other. Encapsulation protects the data in an object from accidental damage, but allows other objects to interact with that data by calling the object's member functions and structures.
- Subclassing and inheritance make it possible to extend and modify objects through deriving new kinds of objects from the standard classes available in the system. Thus, new capabilities are created without having to start from scratch.
- Polymorphism and multiple inheritance make it possible for different programmers to mix and match characteristics of many different classes and create specialized objects that can still work with related objects in predictable ways.
- Class hierarchies and containment hierarchies provide a flexible mechanism for modeling real-world objects and the relationships among them.
- Libraries of reusable classes are useful in many situations, but they also have some limitations. For example:
- Complexity. In a complex system, the class hierarchies for related classes can become extremely confusing, with many dozens or even hundreds of classes.
- Flow of control. A program written with the aid of class libraries is still responsible for the flow of control (i.e., it must control the interactions among all the objects created from a particular library). The programmer has to decide which functions to call at what times for which kinds of objects.
- Duplicahon of effort. Although class libraries allow programmers to use and reuse many small pieces of code, each programmer puts those pieces together in a different way. Two different programmers can use the same set of class libraries to write two programs that do exactly the same thing but whose internal structure (i.e., design) may be quite different, depending on hundreds of small decisions each programmer makes along the way. Inevitably, similar pieces of code end up doing similar things in slightly different ways and do not work as well together as they should.

Class libraries are very flexible. As programs grow more complex, more programmers are forced to reinvent basic

solutions to basic problems over and over again. A relatively new extension of the class library concept is to have a framework of class libraries. This framework is more complex and consists of significant collections of collaborating classes that capture both the small scale patterns and major mechanisms that implement the common requirements and design in a specific application domain. They were first developed to free application programmers from the chores involved in displaying menus, windows, dialog boxes, and other standard user interface elements for personal computers.

Frameworks also represent a change in the way programmers think about the interaction between the code they write and code written by others. In the early days of procedural programming, the programmer called libraries provided by the operating system to perform certain tasks, but basically the program executed down the page from start to finish, and the programmer was solely responsible for the flow of control. This was appropriate for printing out paychecks, calculating a mathematical table, or solving other problems with a program that executed in just one way.

The development of graphical user interfaces began to turn this procedural programming arrangement inside out. These interfaces allow the user, rather than program logic, to drive the program and decide when certain actions should be performed. Today, most personal computer software accomplishes this by means of an event loop which monitors the mouse, keyboard, and other sources of external events and calls the appropriate parts of the programmer's code according to actions that the user performs. The programmer no longer determines the order in which events occur. Instead, a program is divided into separate pieces that are called at unpredictable times and in an unpredictable order. By relinquishing control in this way to users, the developer creates a program that is much easier to use. Nevertheless, individual pieces of the program written by the developer still call libraries provided by the operating system to accomplish certain tasks, and the programmer must still determine the flow of control within each piece after it's called by the event loop. Application code still "sits on top of" the system.

Even event loop programs require programmers to write a lot of code that should not need to be written separately for every application. The concept of an application framework carries the event loop concept further. Instead of dealing with all the nuts and bolts of constructing basic menus, windows, and dialog boxes and then making these things all work together, programmers using application frameworks start with working application code and basic user interface elements in place. Subsequently, they build from there by replacing some of the generic capabilities of the framework with the specific capabilities of the intended application.

Application frameworks reduce the total amount of code that a programmer has to write from scratch. However, because the framework is really a generic application that displays windows, supports copy and paste, and so on, the programmer can also relinquish control to a greater degree than event loop programs permit. The framework code takes care of almost all event handling and flow of control, and the programmer's code is called only when the framework needs it (e.g., to create or manipulate a proprietary data structure).

A programmer writing a framework program not only relinquishes control to the user (as is also true for event loop programs), but also relinquishes the detailed flow of control within the program to the framework. This approach allows the creation of more complex systems that work together in interesting ways, as opposed to isolated programs, having custom code, being created over and over again for similar problems.

Thus, as is explained above, a framework basically is a collection of cooperating classes that make up a reusable design solution for a given problem domain. It typically includes objects that provide default behavior (e.g., for menus and windows), and programmers use it by inheriting some of that default behavior and overriding other behavior so that the framework calls application code at the appropriate times.

There are three main differences between frameworks and class libraries:

- Behavior versus protocol Class libraries are essentially collections of behaviors that you can call when you want those individual behaviors in your program. A framework, on the other hand, provides not only behavior but also the protocol or set of rules that govern the ways in which behaviors can be combined, including rules for what a programmer is supposed to provide versus what the framework provides.
- Call versus override. With a class library, the code the programmer instantiates objects and calls their member functions. It's possible to instantiate and call objects in the same way with a framework (i.e., to treat the framework as a class library), but to take full advantage of a framework's reusable design, a programmer typically writes code that overrides and is called by the framework. The framework manages the flow of control among its objects. Writing a program involves dividing responsibilities among the various pieces of software that are called by the framework rather than specifying how the different pieces should work together.
- Implementation versus design. With class libraries, programmers reuse only implementations, whereas with frameworks, they reuse design. A framework embodies the way a family of related programs or pieces of software work. It represents a generic design solution that can be adapted to a variety of specific problems in a given domain. For example, a single framework can embody the way a user interface works, even though two different user interfaces created with the same framework might solve quite different interface problems.

Thus, through the development of frameworks for solutions to various problems and programming tasks, significant reductions in the design and development effort for software can be achieved. A preferred embodiment of the invention utilizes HyperText Markup Language (HTML) to implement documents on the Internet together with a general-purpose secure communication protocol for a transport medium between the client and the merchant. HTTP or other protocols could be readily substituted for HTML without undue experimentation.

Information on these products is available in T. Berners-Lee, D. Connoly, "RFC 1866: Hypertext Markup Language—2.0" (November 1995); and R. Fielding, H, Frystyk, T. Berners-Lee, J. Gettys and J. C. Mogul, "Hypertext Transfer Protocol—HTTP/1.1: HTTP Working Group Internet Draft" (May 2, 1996). HTML is a simple data format used to create hypertext documents that are portable from one platform to another. HTML documents are Standard Generalized Markup Language (SMGL) documents with generic semantics that are appropriate for representing information from a wide range of domains. HTML has been in use by the World-Wide Web global information initiative since 1990. HTML is an application of ISO Standard 8879:1986 Information Processing Text and Office Systems; Standard Generalized Markup Language (SGML).

To date, Web development tools have been limited in their ability to create dynamic Web applications which span from client to server and interoperate with existing computing resources. Until recently, HTML has been the dominant technology used in development of Web-based solutions. However, HTML has proven to be inadequate in the following areas:

Poor performance;

Restricted user interface capabilities;

Can only produce static Web pages;

Lack of interoperability with existing applications and data; and

Inability to scale.

Sun Microsystem's Java language solves many of the client-side problems by:

Improving performance on the client side;

Enabling the creation of dynamic, real-time Web applications; and

Providing the ability to create a wide variety of user interface components.

Java is compiled into bytecodes in an intermediate form instead of machine code (like C, C++, Fortran, etc.). The bytecodes execute on any machine with a bytecode interpreter. Thus, Java applets can run on a variety of client machines, and the bytecodes are compact and designed to transmit efficiently over a network which enhances a preferred embodiment with universal clients and server-centric policies.

With Java, developers can create robust User Interface (UI) components. Custom "widgets" (e.g. real-time stock tickers, animated icons, etc.) can be created, and client-side performance is improved. Unlike HTML, Java supports the notion of client-side validation, offloading appropriate processing onto the client for improved performance. Using the above-mentioned custom UI components, dynamic real-time Web pages can also be created.

Sun's Java language has emerged as an industry-recognized language for "programming the Internet." Sun defines Java as: "a simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded, dynamic, buzzword-compliant, general-purpose programming language. Java supports programming for the Internet in the form of platform-independent Java applets." Java applets are small, specialized applications that comply with Sun's Java Application Programming Interface (API) allowing developers to add "interactive content" to Web documents (e.g. simple animations, page adornments, basic games, etc.). Applets execute within a Java-compatible browser (e.g. Netscape Navigator) by copying code from the server to client. From a language standpoint, Java's core feature set is based on C++. Sun's Java literature states that Java is basically "C++, with extensions from Objective C for more dynamic method resolution".

Another technology that provides similar function to JAVA is provided by Microsoft and ActiveX Technologies, to give developers and Web designers wherewithal to build dynamic content for the Internet and personal computers. ActiveX includes tools for developing animation, 3-D virtual reality, video and other multimedia content. These tools use Internet standards, work on multiple platforms, and are being supported by over 100 companies. The group's building blocks are called ActiveX Controls, small, fast components that enable developers to embed parts of software in hypertext markup language (HTML) pages. ActiveX Controls work with a variety of programming languages including Microsoft Visual C++, Borland Delphi, Microsoft Visual Basic programming system and, in the future, Microsoft's development tool for Java, code named "Jakarta." ActiveX Technologies also includes ActiveX Server Framework, allowing developers to create server applications. One of ordinary skill in the art readily recognizes that ActiveX could be substituted for JAVA without undue experimentation to practice the invention.

A preferred embodiment provides a system for building manageable applications. The applications can be readily deployed, on to a variety of platforms, and configured so that it's easy to partition them on to clients versus servers and administer the applications. A preferred embodiment is enabled as a client-server application that's distributed nodes in a multi-node platform. A single application is divided up into pieces and distributed across nodes in the network. Thus, an application is defined as a distributed system.

Enterprise computing systems are often a heterogeneous collection of nodes interconnected by a network. In a typical environment, a server node in the network holds data and programs that interact with the database, and a client node contains a program or programs for accessing the information on the server. The complexity of these environments makes it difficult to create, configure, deploy and administer software applications. However, the advent of Web technologies (browsers, the Java language and HTTP) has enabled enterprises to create and use internal Webs to assist in solving some of these problems. Java enables the Web as a client-server application platform and distribution channel to interact with a preferred embodiment in addressing many of the aforementioned challenges.

A preferred embodiment includes a toolkit for creating client programs that can be downloaded from the Web; interfaces for enabling a server program to function with a client as a single application; tools for connecting both client programs and server programs with a framework for executing the tools; and tools for installing, deploying and administering applications.

An application created in accordance with a preferred embodiment consists of a set of components that cooperate with each other. A server component can be implemented in any source language that can call a C program. A client component is implemented in the Java programming language. The client component consists of a Graphical User Interface (GUI) and a Presentation Engine (PE). To complete the application system, a preferred embodiment provides a communication layer that enables the exchange of messages between the client and the server components, an Exception layer for reporting errors, and an Access layer for managing application deployment. The task of an application developer utilizing a preferred embodiment is to assemble the components into an application system.

A preferred embodiment provides Java client access to server applications and can be utilized to create new applications or extend existing applications. Among existing applications, those that are partitioned so that server programs and databases are on the server, and user interfaces are on the client are especially suited for migration to the preferred embodiment. Other applications especially suited to a preferred embodiment require access to departmental or corporate data without changes to databases or the programs that access them. For enterprises already using the web technologies in an Internet, Intranet or other network environment, applications in accordance with a preferred embodiment provide quick access to existing data from anywhere in the organization or the world. Applications in

accordance with a preferred embodiment can execute on any processor with the Java interpreter/runtime (Java JDK) or JavaOS installed. Application client programs are developed in Java using a null application template that contains the necessary Java classes and methods for integration with a Graphical User Interface (GUI). The template includes Java classes which will allow the client program to communicate with the server program. Scripts and tools for installing and deploying applications, include a generalized startup applet for application launch from a Web browser or applet viewer.

The primary development task in accordance with a preferred embodiment is to create an application front end referred to as a Presentation Engine, hereinafter (PE), from a provided template. The PE template includes methods (logic in an object) to send messages and their data through a client communication library. Developers modify the template to specify the messages and data required for their application. The communication library handles the passing of messages and data across the network. The PE toolkit supports these tasks. The toolkit is a full set of Java components which can be modified and extended for a particular application's requirements. The server program must also be enabled to communicate with the client by specifying which handler functions are called in response to inbound messages and inserting "send" calls to a server-local communication library when data are sent to a client. The two PE development tasks are to: (1) connect a Java UI to the PE framework, and (2) define/describe the messages to be exchanged between the PE and the server.

All of the components in accordance with a preferred embodiment reside on the server so that the server controls all access to its resources. The server thus controls deployment and administration of its set of applications. The clients use the web to access the server's resources. A template is also provided for creating an applet that enables users at client nodes to start applications from a browser which will be discussed in added detail below.

FIG. **4** is a block diagram in accordance with a preferred embodiment in an Enterprise Network **450**. The front end (client) **410** and back end (server) **400** create and receive data messages communicated via messages at their local communication libraries **420**. Messages are events which are encoded in a protocol, "ictp" layered on top of Transmission Control Protocol/Internet Protocol (TCP/IP.) The execution framework **430** layer could be implemented on any network protocol as asynchronous, event-driven message passing on top of a communication protocol, such as TCP/IP, avoiding the dependencies of any particular server protocol. An application has a specific set of message events, and each component (front **410** and back **400** ends) includes local handlers for the set of message events. Thus, any two components can be plugged into the execution framework **430** to form an application **440** if they include local handlers for messages in the set of message events defined for the application **440**. The components each include a local instance of a communication library **420**. A component only interacts with the Application Programming Interface (API) of its local communication library **420** in order to send or receive message events.

Application components in accordance with a preferred embodiment are "loosely coupled" because there is no direct dependence (or communication) between the components. The execution framework **430** provides a consistent connection for all message transfer for the various applications. The execution framework **430** supplies additional services, such as error reporting, for the connection. An application **440** is insulated from the physical properties of the specific distributed platform on which the application is deployed. Thus, the application components and set of message events remain constant while the distributed platform can be scaled and modified.

Since applications **440** are systems containing well defined components as is shown in FIGS. **4** and **5**, it is possible to install sets of front ends **530** and back ends **540** plus the definitions of how to associate them into applications **440**. The server node **520**, therefore, holds all the resources while the client node **500** can only request access to the resources. The server node **520** is extended to allow it to control access to its resources by enforcing well defined policies.

FIG. **5** illustrates how a preferred embodiment leverages Java to facilitate the establishment and implementation of server-centric policies. The client **500** and server node **500** communicate utilizing the web technologies in an Internet, Intranet or other network environment. The client node **500** contacts the server node **520** via HTTP with a request to execute an application. After authenticating the client node **500**, the server node **520** selects front end **502** and back end **510** components based on the application definition list maintained at the server node **520**. The server node **520** starts its selected back end process **510** and sends the selected front end program **502** to the client node **500** via the Web technologies in an Internet, Intranet or other network environment. The client node **500** executes the selected front end **502** locally at the client node **500**. The front end (client) programs **502** open a TCP/IP connection back to the server node **520** to initiate message passing in order to run the applications. The selected front end program **502** is implemented entirely in Java which facilitates instances of client/server applications which can run concurrently on a set of multi-platform clients nodes. The server node **520** is able to send a selected front end program **502** to any client node **500** which has the Java runtime installed on the computer. Server policies will not involve the clients. The policies will focus on the server's control of its local resources.

FIG. **5** also illustrates the loose coupling between the selected front end **502** and the selected back end **510** application components. Each program includes a local communication library **515**. The front end library is implemented in Java, the back end library is implemented in C++ with a C API. The programs each utilize their local communication library API to send and receive messages. Thus, the programs do not communicate directly with each other.

PRESENTATION ENGINE

There are two phases for each application execution. In the first phase, application startup, the client node requests access to the server node's resources and the server acts on this request. The nodes are associated via the Internet or other communication network, and thus do not have a permanent relationship within the enterprise. In the second phase, application execution, the client has received a front end Java program called a Presentation Engine (PE) to facilitate presentation services. The front end and back end are communicating via the execution framework over TCP/IP.

FIG. **6** illustrates the processing associated with application startup in accordance with a preferred embodiment. When an application is started, the client node **600** executes a startup applet **620** which first collects information about the client **600** user and contacts the server node **610** via HTTP **602**. The server node **610** has been extended to include a web server **630** for processing requests via HTTP **602** over the Web technologies in an Internet, Intranet or

other network environment. The access layer **640** is called via a cgi-bin interface from the web server **630**. The access layer **640** provides a framework so that the information about the client node **610** user, for example userid and password, can be used to call server-resident authentication services. Should authentication be successful, the access layer **640** uses the application name, which is also supplied by the startup applet **620**, and invokes the app manager **650**. The app manager **650** handles the application definitions installed on the server node **610**. The app manager **650** selects the application back end **660** and initiates the processing. A listener socket is opened on the server node, and the port number is returned to the app manager **650**. The app manager **650** also selects the application Front End (PE) **670**, stored on the server node **610** as a set of Java bytecodes, and creates a selected PE **680** instance which includes the listener socket port number. Application startup ends when the PE **680** instance is downloaded to the client node **600** for execution.

APPLICATION EXECUTION

Now referring to FIG. **7**, when application execution is initiated, the client node **705** begins to interpret the PE **700** it has received from the server node **710**. The PE **700** is a framework (which includes an User Interface (UI) which can include a graphical UI (GUI)) and an instance of ICE-T a communication library **720** implemented in Java. Once it starts up, the PE **700** opens a socket connection to the server node **710** utilizing the server port number it was supplied when the server app manager **650** started the back end process **730**. The back end process **730** on the server node **710** may be a distributed system which encapsulates an interface to a Data Base Management System (DBMS) **712**. The PE **700** focuses solely on presentation services and its own web access capabilities. The ICE-T execution framework **724** specifically links the PE **700** to the back end component **730** using event-driven messages (TCP/IP) **735**. This design preserves modularity between the PE **700** and the back end **730** of an application.

During application execution, the server node **710** ICE-T communication library **720** manages the connection to facilitate policies that maximize access to its resources. For example, a server can be configured for a maximum time to await a transaction response. A timer runs, if it exceeds the maximum time before new client messages are received the server will terminate the client connection and recycle the port. The ICE-T execution framework **724** supplies runtime error reporting services and application execution data which are accessible to system administrators via Web technologies in an Internet, Intranet or other network environment access. When application execution terminates, the relationship between the client node **705** and the server node **710** also terminates. Application startup must occur each time application execution is desired.

A client's presentation engine **700** can be stored on the client node **705** and started dynamically if the presentation engine **700** is cached on the client. The reason for storing the presentation engine in cache is to support mobile (nomadic) computing, performance and to reduce network traffic. This technique would require versioning to assure that the presentation engine **700** is synched with the latest, most current release on the server node **710**.

APPLICATION DEVELOPMENT

The application development process in accordance with a preferred embodiment is a system integration task rather than a third generation language programming task. FIG. **7** illustrates the three fundamental components of an application in accordance with a preferred embodiment. The front end or presentation engine **700** on the client side **700**, the back end **730** on the server node **710**, and the node ICE-T execution framework **724** which facilitates the connection between the front end **700** and the back end **710**. The application development process consists of a plurality of steps. The first step defines the responsibilities of the front end **700** and back end **710** components. A preferred embodiment is designed to facilitate migration of existing client/server applications to function utilizing the Web technologies in an Internet, Intranet or other network environment for communication services. This migration does not require redesigning existing application logic. The process entails the implementation of a new PE front end implemented in the Java language which exchanges messages **735** with the remaining server back end code **730**. FIG. **8** illustrates the migration of an existing client-server application to one supported by a preferred embodiment.

The new Java PE front end **820** is added to the existing application code and will contain the Presentation Layer (UI tier) **800** and some amount of non-database aware logic responsible for manipulating variables as shown by Presentation Layer (UI tier) **810**. All logic which is aware of the database and its data remains in the back end component. The PE **820** is not limited to logic from the existing application. It is a program implemented in Java, and thus can take direct advantage of additional Internet related services within the enterprise intranet. The complexity of the PE is determined by the developer's choice. The new Java front end **830** includes communication library facilities for passing messages there between.

The second step in migrating existing applications to utilize a preferred embodiment is to define the message events exchanged between the PE and the back end. The PE and the back end are loosely coupled and exchange data events. Each event has a unique name and data associated with the event. Message event data items are declared by type, all primitives plus one composite type. Message data types directly map to primitive types in both Java and C. The communication libraries have APIs which can be used by developers to define message events.

The third step is to set up the back end to handle message events. The server resident back end component must handle message events. This processing is enabled by linking in an instance of the communication library with existing server code. The communication library has a C API, so the server code can be implemented in any language which can call C. The existing server code utilizes the communication library API to send and receive messages. The communication library calls handler functions in the server code when message events come in from the front end. The developer will create these handler functions and register them, one handler function per defined message event, using APIs in the communication library.

Once compiled and linked, the back end is a single process which is started by the app manager during application initiation. The interface between the server communication library and the server code is multi-threaded.

The fourth step is to develop a Presentation Engine front end. Java is utilized for this task. Templates and tools are provided to facilitate this task. The detailed tasks associated with facilitating these tasks are described below.

The final step in migrating an existing client server application to a preferred embodiment is to install applica-

tion components on the server node. Both the front end (client) and back-end components are installed on the server node. A Java startup applet enables clients to access the application on the server. A set of tools for application installation and configuration are provided as applications on server nodes in accordance with a preferred embodiment. A Java startup applet template is also provided to facilitate development of the application's startup applet.

PRESENTATION ENGINE DEVELOPMENT

Presentation engine development is the step of developing a new application front-end in Java. To simplify this task and provide application performance and robustness guarantees, all application front ends are instances of a single class of programs. A basic presentation engine framework implemented in Java is provided as an example in accordance with a preferred embodiment. Developing a specific application presentation engine means extending and customizing the basic presentation engine framework template. A Presentation Engine (PE) is itself a system with two components: (1) a UI implemented in Java, and (2) the PE framework. Developing a PE is a system integration task with the following two steps.

(1) Develop a UI which is Implemented with Java.

FIG. **9** is a block diagram illustrating a PE **960** in accordance with a preferred embodiment. The UI **900** component is developed according to a developer's particular requirements. Since the front end processing utilizes the Java language, the UI also takes advantage of Java's rich functionality. The UI **900** is connected to the PE **960** framework via an interface to the UI Adaptor **9 10** component. The interface is a general message-passing interface that facilitates the attachment of any Java implemented UI **900** with the PE **960** framework to exchange data events. An API is provided to the PE **960** framework so that a developer can link the UI **900** to the UI Adaptor **910**.

(2) Extend the PE Framework Template (**960**)

FIG. **9** illustrates how the PE Framework **960** architecture is arranged to facilitate two kinds of external events in accordance with a preferred embodiment. The two kinds of external events are UI events and Message events from the Execution Framework. The framework includes a Java implemented communication library component, the Comm Adaptor **950**. The PE framework **960** interfaces to the UI **900** via the UI Adaptor **910** and interfaces to the server TCP/IP connection via the Comm Adaptor **950**. The mediator **920** component handles all external events moving into and out of the PE framework **960**.

The UI **900**, UI Adaptor **910** and the Model **940** are directly extended by the developer supplying application specific information. The communication library and mediator components are not extended by developers. The PE framework **960** has a model **940**-view **930**-controller architecture. A developer may extend the Model **940** component in order to implement local logic and maintain PE-local state. The view component **930** maps between the Model **940** data representation and the message data representations.

It is possible to create a PE framework without extending the Model **940** component of the PE Framework **960**. This model is referred to as a "PE-lite" option and consists of the comm adaptor **950**, mediator **920**, UI Adaptor **910** and the UI **900**. To utilize this option, a developer assembles the components from a UI **900** and a PE framework instance **960**. The PE framework **960** makes use of the Mediator **920** to directly map between UI external events and Comm external events.

The execution framework **960** is a communication abstraction that utilizes TCP/IP as a communication backbone, and is analogous to a bus in a computer system. Each component in accordance with a preferred embodiment plugs into the execution framework much as various function cards plug into the bus of a computer system. There is no notion of local versus remote in the components which is unlike a typical distributed computing environment.

In this architecture, there is no distributed domain. Everything is local and the way that a component connects up with this framework is by making local calls to a communication layer which is the framework. It's all embodied in the communication library components. So, the messages that are exchanged between programs are data. Rather than function calls, they are actually data containers. All that can happen in the exchange is that data are packed up and sent as an event, or data are packed up here and sent over there as an event.

Inside of the client and server programs, there are handlers for the data in the message events. So, any front end and any back end has local handlers for a common set of message events. They can be plugged together by the framework to create an application. So, what we call these is, "loosely coupled", which means that there are no dependencies between the programs and they do not communicate directly. This preserves modularity in the application. A program may send data events, and the program also is enabled to handle data events which interrupt its processing.

The framework currently is layered over TCP/IP for a communication vehicle, but other networking protocols can be utilized. The programs always interact with the ICE-T Comm Layer. There is a firewall between the actual, physical distributed platform, underneath and the application above because the application components are interfacing with the abstraction. Thus, applications are independent from the details of the physical structure of the distributed enterprise. The enterprise may be scaled, and/or modified without requiring re-design of the applications.

Example In Accordance With A Preferred Embodiment

FIG. **10** is a block diagram of a prior art client server architecture. In the prior art a client computer **1000** would access a network **1030** via a predefined data protocol (datastream) and interact with a predefined server **1001** containing code **1010** and data in the form of a DataBase Management System (DBMS) **1020**.

An application in accordance with an alternate embodiment is shown in FIG. **11**. The encapsulated DBMS **1102** is conserved, as is the enormous investment in code **1170**. However, the server **1180** includes a service API **1160** for communicating to the network support **1150** which is responsible for turning the web node into a loosely coupled client **1120** utilizing a small amount of Java code and conforming to a predefined datastream **1130** that is sent by the server **1180** to the client **1120**. A web server **1106** for receiving HTTP commands **1110** and transmitting Java bytecodes **1112** is utilized to facilitate access to and transmission of messages over the web from the server **1180** via the access module **1108** and the PE framework **1104** as discussed earlier and reiterated below.

FIG. **12** illustrates a server **1210** establishing contact with a client **1200** through a service API **1200** in accordance with an alternate embodiment. The code **1240** and DBMS **1230** are as described in FIGS. **10** and **11**. However, web contact is established between the nodes using HTTP commands **1260** by authenticating a user utilizing the Access Layer

1280 to establish client 1200—server 1210 communication. Then, the Java bytecodes 1250 are extracted from the PE Repository 1290 and downloaded to the client 1200 using the Access Layer 1280 and the Web Server 1270.

FIG. 13 illustrates a loosely coupled client—server application in accordance with an alternate embodiment. The Server 1300 communicates via a service API 1360 and Network Support 1340 to a Client 1330 utilizing a predefined data stream 1350 and application code 1320 and a DBMS 1310. The "loosely" coupled nature of the application architecture is enabled through message communication in an asynchronous manner letting the Client 1330 and the Server 1300 maintain state independence and connect via a low bandwidth network 1350. The applications require no function call level API's.

Some of the basic architectural design decisions that form the foundation of a preferred embodiment include the Server 1300 controlling both the Client 1330 and Server 1300 states. The Java language is utilized to communicate via the Web technologies in an Internet, Intranet or other network environment to distribute Client 1330 state information to the Client node. The Server 1300 presents an encapsulated DBMS interface to the network. The Server 1300 node is extended by a framework to support the establishment of a Client 1330—Server 1300 relationship for web nodes.

The Client 1330—Server 1300 relationship for web nodes is established utilizing a secure http process which authenticates the user at the requesting node. Then, the Client 1330 node is established by selecting an appropriate Client 1330 state from stored states at the Server, and the Java client state is downloaded over the network to the particular Client 1330 node. Next, the Client 1330—Server 1300 network communication session is commenced by starting the server process and establishing a socket connection to the Client 1330 node. Once the session is established, then the network must be managed by maintaining event-driven message passing with the PE communication library.

FIG. 14 illustrates an execution framework 1440 and the system integration task necessary to develop an application 1430 in accordance with a preferred embodiment. First, the developer defines the message events 1450 that must be exchanged between components. The client program 1410 and server program 1420 are defined as applications 1430 on the Server Node 1420. The back end must be enabled to send and receive messages. The front end (PE) must be developed, then the components are installed in the server node 1420. The goal is to support modular application development and require minimal additional knowledge burden (e.g. sophistication in Java or OOP) from developers.

FIG. 15 is a block diagram illustrating the modular design of a client application (PE) in accordance with a preferred embodiment. The application logic 1530 is written entirely in Java to maximize access to Internet facilities. The application logic runs as a Java program or an applet. An External Interface Library 1520 contains various callable functions for interfacing to the UI or API 1500 through the UI layer 1510. A Message Library 1540 provides messages for use in communicating information through the communication layer 1550 to the Server 1560. The architecture is a client loosely coupled with the server 1560. Data exchange is via event-driven message passing. A flexible UI 1510 and communication layer 1550 facilitates message passing APIs interfacing to a Server or UI designs. Developers are not required to understand network communication or system programming in order to design and implement applications. The client program can be designed to support efficient presentation services. The DBMS interface is encapsulated in the back end (server). Thus, there are no distributed transactions which must be handled in the web connection between client and server.

FIG. 16 is a block diagram of a framework in accordance with an alternate embodiment. Model data 1600 is provided in the form of predefined classes that can be accessed and modified by a user to add additional data or logic to meet the requirements of a particular application and specify appropriate maps for views or UI components. All of the class templates necessary to be extended into a fully functional model are provided. The Model data 1600 is utilized on the external IF View 1622 and additional model data 1620 may be added as necessary to expand the view. Similarly, the message view 1612 can be expanded by adding additional model data 1610 to the message view 1612. A mediator 1630 is responsible for routing messages and granting control to the Communication Adaptor 1640 or the UI Adaptor 1650.

FIG. 17 illustrates the basic building blocks in accordance with an alternate embodiment. A PE object 1700 is an object containing methods (logic) and data (public and private information) that is utilized to facilitate a reusable, extensible function. A PE function 1710 utilizes PE objects 1700 to implement callbacks, observers, event handlers and maps in accordance with a preferred embodiment. PE Info 1720 is a basic unit of data moved between functional components, and a PE Event Handler 1730 is used by views to handle incoming messages and UI events in accordance with a preferred embodiment.

FIG. 18 is a block diagram highlighting the steps utilized to extend the (PE) framework model 1850 in accordance with a preferred embodiment. A user must modify the UI Adaptor 1800 as shown at 1810. In addition, user input is needed to modify the Comm Layer 1840, the mediator 1860, add to the Custom Comm Mechanism 1870 and the Custom Input Mechanism 1880. All of the aforementioned elements communicate with each other through Mediator 1860. The mediator 1860 communicates, in turn, with Model 1850 through the External IF View 1820, which can be modified via its section 1822, and Message View 1830, which can be modified by its section 1856. All of the base processing necessary to facilitate the application is incorporated into the basic framework which provides all of the basic classes for instantiating objects necessary to perform user's requirements.

FIG. 19 is an illustration of a PE Object 1900 in accordance with a preferred embodiment. The PE object is a data structure library that is used for building the model and other structured data passed around in messages. There is an associated containment hierarchy with primitive 1910 and composite 1920 objects instantiated from the base PE Object 1900. An object-oriented architecture facilitates polymorphic, encapsulated objects that are fully self-describing and fully enabling for a serialization interface.

FIG. 20 is an illustration of a PE Event Handler 2000 used by Views to handle incoming messages and UI events in accordance with a preferred embodiment. The PE Event Handler 2000 dispatches tasks based on a name field and data defined in objects PEInfo 2010 data which is passed utilizing a message to implement a map to the model. FIG. 21 illustrates a PEInfo object 2100 data in accordance with a preferred embodiment. The data encapsulates event and message signatures and is used in three places for passing between model and XIF View, Model and Msg View and XIF View and the UI.

FIG. 22 illustrates incoming message flow to a model in accordance with a preferred embodiment. A message flows

into a communication layer in bytestream format **2200** and passed via a mediator **2202** to a message view **2210** to unpack and dispatch the message utilizing a MsgHandler PE Event Handler to transfer the PEInfo **2220** data. The PEInfo **2220** is used to encapsulate the event and message signatures and transform the message into a model map **2230** which is thereafter utilized for setValue and getValue processing **2240**. When a UI Event, such as a key press or other activity occurs at **2260**, a callback PE Function **2270** utilizes PEInfo **2280** to pass event and data information to PEInfo **2294**. The PEInfo **2294** is used to encapsulate the UI Event signatures and transform them into a model map **2292** which is thereafter used by PEObject **2250** for further message processing the external view dispatcher **2290** for transformation into information that the Model PEObject **2250** can utilize for further message processing.

FIG. **23** illustrates incoming messages mapping a UI to a Model in accordance with a preferred embodiment in a manner very similar to FIG. **22**. A message flows into a communication layer in bytestream format **2300** and passed via a mediator **2302** to a message view **2310** to unpack and dispatch the message utilizing a MsgHandler PE Event Handler to transfer the PEInfo **2320** data. The PEInfo **2330** is used to encapsulate the vent and message signatures and transform the message into a model map **2330** which is thereafter utilized for setValue and getValue processing **2240**. When a UI Event, such as a key press or other activity occurs at **2350**, a callback function **2360** utilizes PEInfo **2370** to pass event and data information to the external view dispatcher **2380** which sends that information, in turn, to PEInfo **2382**. The PEInfo **2382** is used to encapsulate the UI Event signatures and transform them into a model map **2384** which is thereafter used by PEObject **2342** for further message processing. FIG. **24** illustrates outgoing messages mapping a model to messages in accordance with a preferred embodiment in a manner similar to FIGS. **22** and FIG. **23**. In the FIG. **24** arrangement, an outgoing message flows from set Observer **2440** through the PE Primitive object **2445** and essentially reverses the path it took in FIG. **23**, being passed through PEInfo **2430**, Message View **2420**, Monitor **2402** and Common Layer **2460** where it appears as bytestream **2400**. The reverse flow also occurs with respect to a UI at the left side of FIG. **24** where set Observer **2450** passes an event message through PEInfo **2460**, External View **2465**, Monitor **2402**, PEInfo **2470**, UI dispatcher **2480** and UI Model Map **2485** to validate UI values. FIG. **25** illustrates outgoing messages mapping a model to a UI in accordance with a preferred embodiment in a manner similar to the foregoing Figures. In the FIG. **25** arrangement, an outgoing UI message flows from set Observer **2550** through the PE getValue object **2255** and essentially reverses the path it took in FIG. **23**, being passed though PEInfo **2560**, External View **2570**, Monitor **2572**, PEInfo **2580**, UI dispatcher **2590** and UI Model Map **2595** to validate UI values. The reverse flow also occurs with respect to a message at the right side of FIG. **25** where set Observer **2540** passes an event message through PEInfo **2530**, Message View **2520**, Monitor **2572**, PEInfo **2470** and Comm Layer **2510** from which element it exits as message bytestream **2500**. To further clarify processing in accordance with a preferred embodiment, the detailed installation and logic specification for an application in accordance with a preferred embodiment is presented below.

Enterprise computing environments usually consist of many kinds of nodes interconnected by a network. In a typical environment, a server node in the network holds data and programs that interact with the database, and a client node contains a program or programs for accessing the information on the server. The complexity of these environments makes it difficult to create, configure, deploy, and administer software applications. The advent of Web technologies (browsers, the Java language, HTTP) has enabled enterprises to create and use internal Webs to help solve some of these problems. FIG. **26** illustrates an arrangement that facilitates the launch and utilization of an application URL in accordance with a preferred embodiment. When the application URL **2615** is started, the client node **2600**, through its browser **2605**, executes startup applet **2610** to thereby collect information about the client user and contact the server node **2620**. The server node **2620** includes a server program **2625**, a database **2630**, and access layer **2635**, and application manager **2640**, a web server **2645** and presentation engines **2650**. The details concerning launch and utilization of application URL **2635** can be found in the discussion of FIG. **6** above.

Java enables the Web as a client-server application platform and distribution channel. The "Interprise" Computing Environment Toolkit (ICE-T) enables building, extending and deploying client-server applications for the Web. ("Interprise" combines the internet and the enterprise.)

The ICE-T application provides:

- A toolkit for creating client programs that can be downloaded on the Web
- Interfaces for enabling a server program to work with a client as a single application
- The tools to connect both client program and server programs to a framework for executing them
- Tools for installing, deploying, and administering applications

ICE-T Applications

An application consists of a server program component (implemented in any language that can call C) and a client program component (implemented in Java). The client component consists of a GUI and a Presentation Engine (PE). To complete the application system, ICE-T provides a Communication Layer that enables the exchange of messages between the client and server components, an Exception Layer for error reporting from ICE-T components, and an Access Layer for managing how applications are deployed. The task of the ICE-T application developer is to assemble the components into an application system.

ICE-T provides Java client access to server applications. New applications or existing ones extended utilizing a preferred embodiment. Among existing applications, those that are partitioned so that server programs (business logic) and databases are on the server, and user interfaces are on the client (three-tier client-server applications) are especially suited for migration to ICE-T front ends.

ICE-T is especially suited for enterprise applications that require:

- Access to department or corporate data without changes to databases or the programs that access them.
- For enterprises already using an enterprise web technologies in an Internet, Intranet or other network environment, ICE-T applications can provide quick access to existing data from anywhere in the organization or the field where there is an internet connection.
- Client platform independence.
- ICE-T Presentation Engines can run anywhere the Java Virtual Machine is present.

Rapid development.

Clients are developed in Java using a null application template that contains the necessary Java classes and methods for integration with a GUI and a Communication Library.

Easy deployment.

ICE-T provides scripts and tools for installing and deploying applications, include a generalized startup applet for providing application launch from a Web browser or applet viewer.

Centralized authentication of users.

A customizable Access Layer, installed on the server, enables centralized control of access to client programs.

Easy maintenance.

For most enterprises, maintaining existing applications is a tremendous resource burden. ICE-T provides the means to make new application front-ends (clients), or migrate existing ones, without changing the architecture or programs of the back-end (server), or requiring a steep learning curve.

Wide use throughout the enterprise, from the desktop or the laptop. A familiar and pervasive interface.

End-users can name ICE-T applications as applets in a Web browser.

ICE-T Application Development and Deployment

The primary development task in an ICE-T application is to create an application front end, a Java Presentation Engine, from the provided template. The Presentation Engine (PE) template includes methods to send messages and their data through a client Communication Library. Developers modify the template to specify the messages and data required for their application. The Communication Library handles the passing of messages and data across the network. The Presentation Engine Toolkit supports these tasks. The Toolkit is a full set of Java components some of which you must modify or extend.

An additional development task is to modify the server program to specify which function to call in response to inbound messages and make calls to a server Communication Library to send results to the client. All of the components in an ICE-T application system reside on the server. To deploy an application, you install its components and additional ICE-T files and programs that manage applications on the server. ICE-T also provides a template for creating a startup applet that enables users to start applications from a browser. Chapter 3, "Configuring and Deploying ICE-T Applications" describes these tasks and tools.

Event-Driven Message Passing in ICE-T Applications

The components of ICE-T applications create, send, and receive messages in response to external events. An event is a way of communicating that something has happened, such as input from the user (a mouse click) or a change in the system environment (a server shutdown).

The ICE-T Communication Layer enables asynchronous event-driven message passing between client and server program components on TCP/IP. In ICE-T, the messages themselves are events, and each program component includes local handlers for message events.

Client and server program components must each be set up to create, send, and receive messages. Part of the Presentation Engine's functionality is to determine the recipients and deliver the messages to them. For example, the user interface sends a message to the Presentation Engine when a user clicks a button. The Presentation Engine receives the message and then either sends it to the data model or the server program, which then performs an operation and replies with the result.

The server program must also be able to receive messages and must register functions to handle them.

The recipients of messages do not need to know much about each other, they just need to specify what messages they want to receive. This information is registered using the createMessageHandlers ( ) method in the client program and the createMessageHandlers ( ) function in the server program.

Presentation Engines should include handlers for two kinds of application events:

Events from the user interface

Events coming in to the Presentation Engine from the user interface result in messages to the server or the Presentation Engine's data model.

Events from the server

Events coming in to the Presentation Engine from the server result in displaying data in the user interface or putting data in the Presentation Engine's data model.

Server programs should include handlers for messages from the client. Typically, the handler would call application logic and send the resulting message and it's data back to the client through the server Communication Library.

ICE-T Application Execution

ICE-T applications are designed to work within existing client-server environments. They differ from familiar client-server applications in some key ways.

Typically client programs are developed, maintained, and distributed periodically, taking long cycles of development time, and requiring time to deploy to client nodes. Users who are not on a node on the distribution route may miss software updates. Development can consume resources because of the difficulty of the language or tools used.

Compiled ICE-T Presentation Engines are installed on the server and downloaded on request through HTTP servers. A pair of Communication Libraries behave as a framework for executing the application. This communication layer handles the marshaling and unmarshalling of the message data transferred between client and server. Both client and server should be prepared to handle shutdown events. ICE-T provides default shutdown handlers for this purpose, and developers can add their own.

How ICE-T Applications Work

Before developing an ICE-T application, you might find it useful to see how ICE-T applications work, from both an end-user's perspective and inside ICE-T.

The User View

ICE-T applications can use a Java-enabled Web browser for client access to application execution. Although developers may choose to have applications launched outside a browser, a Web page presents a familiar and easy to use interface for launching applications.

The user begins by opening a Web page and clicking on the URL for the application she wants to run. The URL is the address for a Web page that includes an ICE-T application

startup applet. The Web page with the startup applet is loaded into the browser. The applet collects access information from the user. The applet contains the URL of the server holding the application components and the application name. This information is processed on the server. If the user name, password, and chosen application are authorized, the server downloads a Presentation Engine to the user's node.

The Presentation Engine presents the user with an interface for interacting with the server program and data. It also determines where to pass messages for handling, either to the server program or to its own data model. One example of a client program is one that communicates with a server program that searches a database, for example an employee database. Such a client program might have these user interface elements:

A text field where users enter a first or last name that they wish to find in the employee names database

Buttons for clearing the fields (Clear), exiting the application (Quit), and launching the search (Search)

A scrolling window for viewing the results of the query

The user enters a first or last name and presses Return or clicks a Search button. The client program sends the query to the server, where the server program searches the employee database for a matching name. If a match is found, the server returns the results to the client. The results are displayed in a window on the client.

The ICE-T View

When a user launches an ICE-T application, the client node establishes a Web connection with the server node using HTTP. The server manages this Web connection. ICE-T applications can be launched from a browser, an applet viewer, or as standalone applications. FIG. 26 illustrates the steps associated with launching an application URL in accordance with a preferred embodiment. On the server side, the ICE-T Access Layer (a cgi-bin executable) authenticates the user data. If the authentication succeeds, the Access Layer contacts the ICE-T Application Manager and the Application Manager starts the server program and initiates a network session.

The Application Manager downloads an HTML page with a startup applet for the application. When the user runs the startup applet, the Application Manager selects a compiled Presentation Engine and downloads an HTML page containing the applet tag for it to the client using HTTP. The compiled Presentation Engine includes a GUI and an instance of the client Communication Library and is ready for execution in a Java-enabled browser or anywhere the Java Virtual Machine is installed.

The client node then executes the Presentation Engine locally. The Presentation Engine makes a TCP/IP connection to the server where the server program is running, and the client and server programs cooperate to execute the application.

When a user interface event occurs—for example, when the user enters a first name in the text field and clicks a Search button—the user interface passes a message to the Presentation Engine. The Presentation Engine either sends the message to its data model for processing by the client, or passes the message to the server for processing by the server program. The Presentation Engine determines where on the client a message is handled based on how you have registered message handlers. When the server program sends a message with data to the client, the Presentation Engine displays the result.

The exchange of messages between client and server is handled through the ICE-T Communication Libraries in the ICE-T Communication Layer. When the client program terminates, the Application Manager closes the socket connection to the client and terminates any server processes it started.

ICE-T Task Summary—Building Program Components

ICE-T's modular architecture makes it easy to distribute development tasks to multiple developers. Alternatively, a single developer can complete ICE-T development tasks in stages. Developing a client program requires making a Java Presentation Engine, connecting it to a user interface and to the ICE-T Communication Layer so that it can communicate with a server program The server program could be an existing program or a new one. It must be written in a language that can call C so that it can work with the ICE-T Communication Layer. You don't need to develop new server programs for ICE-T applications, but you must enable the server program to handle messages from the client.

ICE-T Application Building Blocks

An ICE-T application consists of a client program and a server program communicating through a Communication Layer. The client program consists of:

A GUI built with Java

A Java Presentation Engine built using a template.

These components, and related classes used by the Presentation Engine, combine to behave as a single client under the control of the Presentation Engine. The Presentation Engine may be presented as an applet launched from an applet viewer or a Web browser, or as a standalone application.

The server program, new or existing, is developed however the developer chooses. It must be in a language that calls C and it must include functions for handling messages from the client. A template with these functions is provided, as is a main routine that makes calls to the provided server Communication Library.

ICE-T provides these templates, tools, and libraries for developing applications:

pe_template.java

A template for a working Presentation Engine.

ICE-T packages (supplementary to the standard Java packages)

server-template.c and server_template.cc.

Server program templates (one each for C and C++) that define and enable message passing to and from the client. The templates can be used with existing programs or used as a starting point for developing server programs. These templates are analogous to the pe_template used for the client.

ICE-T message data types that work the same on both client and server.

ICE-T also provides a framework that the developer does not modify and in which ICE-T applications can execute. This framework includes:

Communication Layer

Supports network communication between client and server programs. The server Communication Library presents a C API to the server program, which is linked to this library. The client Communication Library is implemented in Java.

ICE-T Exception Layer

Provides exception reporting from the ICE-T application (client and server) in addition to standard system error reporting.

The Presentation Engine

Every Presentation Engine extends (inherits from) a Java class named PresentationEngine. All of the objects that the client program needs are either in the Presentation Engine class or called by it. ICE-T provides a class that extends java, and calls the methods that you need to create a working Presentation Engine.

The filename for the Presentation Engine template is pe_template.java. You can find it in the ICE-T application installation directory under Templates/C or Templates/C++. The file is placed in each of the Template subdirectories for convenience. The pe_template is the same in both files.

FIG. 27 is a table which describes the forms of a Presentation Engine, as an abstract Java class, a template for development, and an executable component in an application in accordance with a preferred embodiment.

To create a working Presentation Engine, you copy the pe_template file and make these changes:

Supply your own Presentation Engine name.

Create user interface components or map the ones from a Java GUI builder.

Create a data model (if your application requires client-side processing).

Define messages and their handlers.

Register message handlers.

These steps are described in this chapter.

Presentation Engine Implementation Options

Developers can create Presentation Engines that simply send messages from the user interface to the server and display the data that the server returns on the client. This option for Presentation Engine implementation is called a "PELite." An alternative implementation of Presentation Engine handles some local data storage in what is called the Presentation Engine's data model. For the latter, developers are required to implement a createModel( ) method, described in this chapter. Both options are supported by the pe_template.

ICE-T Classes and Packages

The documentation for the ICE-T Presentation Engine API is presented in HTML form, like the Java API documentation, and is accessible from the following URL:
file :///<ICE-T Installation Directory>/doc/api
where <ICE-T Installation Directory> is the name of the directory on your system where you have installed ICE-T. The C and C++ header files used by the server program are in the ICE-T installation directory under Server.

ICE-T Message Data Types

ICE-T message data types:

Can be used to construct the data model in the Presentation Engine

Are available for use in the server program for use in application logic

Have analogs on both the client (Java) and server (C)

Use basically the same APIs on both the client and server

Can contain only other ICE-T data types

Are used in messages from client to server, and server to client, where they will appear as a local data structure of their analogous type.

The table appearing below describes the primitive ICE-T message data types and their analogous types on client and server. The prefixes Pe and Srv are provided for your convenience in naming things. They are handled as the same type by ICE-T.

ICE-T Message Types (Primitive)

Data Type	Client (PE)	Server (C)
Petbar, SrvChar	char (Unicode) 16 bits]	char (C) 8 bits
PeString, SrvString	string	char*
PeBoolean, SrvBoolean	Boolean 16 bits	int 1 byte
PeInteger, SrvInteger	int 32 bits	int 32 bits
PeLong, SrvLong	long 64 bits	long 64 bits
PeFloat, StyFloat	float 32 bits	float 32 bits
PeDouble, SrvDouble	double 64 bits	double 64 bits

1. ICE-T transmits only the ASCII (8 bits).

ICE-T supports a composite data type based on the Java vector class (java). util.Vector). It differs from the Java vector type in that it provides an implementation that works on both sides of the ICE-T application (Java clients and C or C++ server programs), and it only takes ICE-T primitive data types.

TABLE

ICE-T Vector Type

Data Type	Description
PeVector, StyVector	A collection that implements a variable-length array. Add elements sequentially with the add Element ( ) method. Access an element by its index.

Working with the ICE-T Directory Structure

Before developing applications, copy the provided templates and Makefiles to an application development directory. There are two subdirectories of templates.

<ICE-T Installation Directory>/Templates/C

Contains Example. mk, pe_template.java, and server_template.c.

<ICE-T Installation Directory>/Templates/Cplusplus.

Contains Example .mk, pe_template.java, and server_template. cc.

For example, create an application directory for an application named myAppName in which the server program is written in C++, and copy the templates to it:

% mkdir myAppNams

% clo <ICE-T Installation Directory>/Templates/C++/*<ICE-T Installation

Directory>/Applications/myAppName/.

The ICE-T installation scripts and MakeHierarchy depend on the newly created application directory being two levels below the ICE-T installation directory. If you choose not to follow the directory setup suggested by the example, you will have to supply options or arguments indicating where the ICE-T installation directory is in relationship to the directory you have created.

Designing the Client Program

Here are some design decisions to make before you develop the client program for an ICE-T application.

Using a graphical user interface (GUI) toolkit, specify and design the user interface.

Determine the events the client and server components of the application should handle. Name each of these events and associate data with it. In this step you answer the question of what messages will be passed between the Presentation Engine and the user interface and between the Presentation Engine and server.

Decide what application logic to put in the client program, if any.

Decide what, if any, data processing you want the client program to handle. This affects whether you create event handling for data updates in the Presentation Engine or just use the Presentation Engine as a mediator between the user interface and the server data (a "PE Lite").

Specify how the user interface should be updated when the Presentation Engine receives messages and data from the server.

Specify how the Presentation Engine receives and decodes data from the server and whether it stores the data for later retrieval or displays the data directly.

If you choose to store data in the Presentation Engine, specify how it is to be stored and updated and what other components in the application depend on that data.

Developing the User Interface

You can create the user interface for the ICE-T application at the same time as, or before, you create the Presentation Engine. Even though they can be separate, the Presentation Engine and user interface work together, so the interactions between them should be planned.

Creating a Presentation Engine

Create a Presentation Engine involves the following basic steps that are described in detail throughout this section:

1. If you have not done so already, copy one of the Templates subdirectories to the Applications directory.

The ICE-T installation directory includes a communication directory for developers to use. To create applications, make a subdirectory under applications for each ICE-T application you develop.

2. Modify and rename the pe_plate file to suit the application. Use the same name for the class and the file.

3. Create a Presentation Engine class.

Create your own Presentation Engine definition using pe_template.java.

4. Integrate the Presentation Engine with the user interface (GUI).

Create a separate user interface class using your chosen Java tool. Integrate the GUI with the Presentation Engine by implementing the createUI ( ) method that is found in pe_template.java.

"Working with the ICE-T Directory Structure" describes how to implement createUI( ).

5. Determine and define the message events to be passed between the Presentation Engine and server program and the Presentation Engine and user interface.

Name each of these events and associate data with it. Specify the type of data that will be passed in response to each message event.

a. Add handlers for events on GUI components.

Implement the operations you want the application to perform in response to user interface events.

b. Write the code for handling incoming message events in the Presentation Engine.

Define handlers in the Presentation Engine for incoming messages from the GUI. Define handlers in the Presentation Engine for incoming messages from the server.

Developers can choose to write a separate class, or classes, in separate files to handle events. This means that a named handler can be modified without modifying the Presentation Engine framework itself.

6. Register which handlers will be sent which messages when events occur (map messages to handlers).

Specify what handler should receive each message.

"Registering Handlers" describes this step.

7. Build the Presentation Engine. ICE-T provides a makefile to build both server and client programs.

These activities are analogous to the tasks for modifying server programs to work with clients. In both client and server program cases, ICE-T provides templates with methods (Presentation Engine) and functions (server) for registering handlers. Developers provide only the application-specific messages and data.

Creating a Presentation Engine Class

Copy one of the Templates subdirectories to the Applications subdirectory. There are two subdirectories, one for C and one for C++. For example, create a directory for an application named myAppName. If the server program for myAppname is to be written in C, copy all of the files from the Templates/C directory:

```
% inkair myAppName
% cp <ICE-T Installation Directory>/Templates/C/*
<ICE—T Installation Directory>/Applications/
myAppName/.
```

For each Presentation Engine you create, modify pe_template.java to declare a class that extends the abstract Java class PresentationEngine:

```
public class myPresentationEngine extends Presentation-
  Engine {//methods for your application}
```

Note—Be sure to give the class and the file the same name. For example, if the Presentation Engine class is named myPresentationEnqine, the file should be named myPresentationEnqine.java.

pe_template.java contains the class and method declarations that you implement to provide the functionality you want.

The methods must include:

```
createuI ( )
createModel ( )
createMessageHandlers ( )
initializeApplication ( )
```

Implementing createModel ( ) and initializeApplication ( ) is optional.

You do not have to use initializeApplication ( ) unless your client program requires local initialization before communication is started.

Importing Packages

The pe_template imports the appropriate packages, including these ICE-T packages and standard Java packages:

```
sunsoft.ice.pe
sunsoft.ice.pe
java.net
java.io
java.applet
```

java.util

java.awt

The Java language package Uava.lang) is implicitly used and does not have to be explicitly imported.

To include the required packages, a Presentation Engine class must have these import statements. Don't delete them from the pe_template.

```
import sunsoft. ice.pe.
import java.net.*;
import java.io.;
import java .applet.*;
import java.util.*;
import java.awt.*;
```

Integrating the User Interface with the Presentation Engine

ICE-T applications must make user interface elements from the application GUI available as components that the Presentation Engine can access. The Presentation Engine reads and handles data from events in the user interface and updates the user interface with data returned from the server.

Creating a GUI that is separate from the Presentation Engine, offers developers more flexibility because updates to the GUI do not require major changes to the Presentation Engine. Use the createUI ( ) method to make the GUI components available to the Presentation Engine.

To implement the createuI ( ) method with application-specific operations, write a body for the method in which you:

Declare variables for the user interface elements

Create a frame to hold the GUI applet (optional)

instantiate and initialize the GUI applet

Get the user interface components

Provide initial values for fields in the GUI (optional)

Add the GUI elements to a Container object (provided in the PresentationEngine class)

For example, here is excerpted code from a createUI ( ) method that declares variables for labels and text fields, creates a frame, and adds the user interface elements to a container component defined in the PresentationEngine class.

```
protected void createUI ( ) {
    Label label_1, label_2, label_3;
    TextField textField_1, textField_2, textField_3;
    //... additional variables
        fr = new Frame ("My Application Title");
        fr.add ("Center", ui);
        fr.pack ( ); fr.show( );
//...
        textField_1.setText (" ");
        textField_2.setText (" ");
//...
        uiContainer.addObject (" firstName", textField_1);
        uiContainer.addObject (" lastName", textField_2);
//...
}
```

Creating a Data Model in the Presentation Engine (Optional)

Updates to the client program can be sent to the user interface or to the data model in the Presentation Engine. If your application is designed to hold data in the Presentation Engine, you use the createModel ( ) method to create the data objects (Observables) to hold the data and attach Observer objects to them.

createModel ( ) is optional and is commented out in the pe_template. To use createModel ( ):

Uncomment the method in the Pe_template

Create Observable data objects to hold data from the server

Attach Observers to the data objects, if necessary

Put the Observable data objects into the data model.

This example instantiates a PeVector data object named empvector. Using the Observer and Observable methods is described in "Attaching Observers to Data Objects."

```
protected void createModel ( ) {
    //create the observable
    PeVector empVector = new PeVector( );
    //Attach observer to the observable
    empVector. addObserver (new empObserver (model));
    //put the observables into Model
    model. addObservable ("empVector", empVector);
}
```

Attaching Observers to Data Objects

Developers have the option of defining any data objects in the model as a simple PeObject (a data item that is sent in messages) or as a PeObservable, a data item with a method associated with it. A PeObservable is a data object with an associated PeObserver function that is called when the data is updated. If a data object changes state, and you want the application to notify any program components that depend on that data, you define each data object as a PeObservable and you attach a PeObserver to it.

ICE-T provides the peobservable and PeObserver classes. Developers call the methods in these classes to add observable data objects to the model and add heir observers to an observer list.

Handling Events

ICE-T client programs receive messages in response to external events. Client program developers need to specify what messages they want the Presentation Engine to receive and register this information using the createMessageHandlers ( ) method. Server program developers have the analogous task of defining message handlers and registering their handlers.

ICE-T provides a layer of abstraction to handle interactions between the GUI and the Presentation Engine. This layer is called the UI Adaptor and it is a set of Java classes that developers use as given. The Presentation Engine makes calls to the UI Adaptor to register messages. The Pe_template includes the necessary methods; developers provide the message bodies as described in the procedures in this chapter.

Presentation Engines handle two kinds of application events:

Events from the user interface

Events coming in to the Presentation Engine from the user interface result in messages to the server or the Presentation Engine's data model.

Events from the server

Events coming into the Presentation Engine from the server result in displaying data in the user interface or putting data in the Presentation Engine's data model.

Handling Events from the User Interface

Applications handle user interface events, such as button clicks and text entry. Java provides an action ( ) method for associating actions with events, such as those on user interface components.

```
public boolean action(Event evt, Object arg) {
//...
}
```

Typical behavior for an action ( ) method is to get the data from the user interface event and send a message to the component of an application that is designated to handle it. In an ICE-T application, those messages are sent to the Presentation Engine using the sendMessage ( ) method.

Here is an example of an action ( ) method when in a GUI. It handles an event in which the user clicks a "Search" button after entering a text string

```
public boolean action(Event evt, Object arg) {
    //. . .
      if (arg. equals ("Search"))
      {
      System.out.println("Search event is detected"); PeString firstName
      = new Pestring (entry_1.getText ( )( );
      PeMessage msg = new PeMessage ("Search");
      msg. addDataElement (firstName);
      PeDebug.println("=====> msg is: "+msg);
      //send this event to the UI adaptor
      pc.sendMessage (msg);
    }
    return true;
}
```

The action ( ) method sends the message defined in:

PeMessage msg=new PeMessage("Search");

The PeMessage class encapsulates the message name and the data. The data in this message, added by the addDataElement ( ) method, is a first name. The sendMessage ( ) method sends the message and its data to the Presentation Engine through the PeUIAdaptor, a class that ICE-T provides for handling communication between the user interface and the Presentation Engine. Developers do not need to modify PeUIAdaptor, but do call methods in it as directed by the pe_template.

Handling Events from the Server

Events from the server program that generate messages for the client can be handled by:

Updates to the user interface

Updates to the Presentation Engine data model.

Write a handler for each event from the server. Include it in the same file as the Presentation Engine, or a separate class file. If you use a separate file for the handler, remember to define it with public access.

Creating Handlers for Updates to the User Interface

To create a handler that updates the user interface, use the template to define a class that extends PeUIHandler. PeUIHandler is an abstract class for a handler of messages that are sent to the user interface.

The constructor method in PeUIHandler provides access to the uiContainer for updating the user interface and for sending messages using the uiadaptor.

This example from the pe_template shows the definition for a class named SampleUIHandler. The constructor method passes adaptor and uiContainer as arguments to give your execute method access to them.

Use this definition as is, but supply the name of the class and add the body of the execute method:

```
class SampleUIHandler extends PeUIHandler {
public SampleUIHandler(PeUIAdaptor adaptor, PeUI uiContainer) {
    super (adaptor, uiContainer);
}
public boolean execute (Object caller, PeMessage message) {
//decode the record sent by the server
//update the ui
return true;
}
```

To enable the Presentation Engine to know what messages to send to SampleUIHandler, register the name of the handler and the data it handles. This step is described in "Registering Handlers for Updates to the User Interface".

Creating Handlers for Updates to the Model

Messages from the server can be handled with code that updates the GUI (as in the example above), or code that stores the data in the Presentation Engine. If you used the createModel ( ) method, you have to create handlers for the events that affect the model.

To create a handler that updates the model, use the template to define a class that:

Extends PeModelHandler.

PeModelHandler is an abstract class for a handler of messages that are sent to the data model you defined with createModel ( ).

Uses the constructor of peModelHandler as is, except to use the new class name in place of PeModelHandler.

public PeModelHandler(PeModel model) { }

This constructor passes the model class as an argument so that your Presentation Engine has access to the model class for updates.

This example from the pe_template shows the definition for a class named SampleModelHandler. Use this definition as is, but supply the name of the class and add the body of the execute method:

```
class SampleModelHandler extends PeModelHandler {
    public SampleModelHandler (PeModel model) {
        super (model);
    }
    public boolean execute (Object caller, PeMessage message) {
    //Application code that decodes the record sent by the server
        //Application code that updates the model
        return true;
    }
}
```

Registering Handlers

Register what component handles what messages in the Presentation Engine by filling in the createMessageHandler ( ) method.

createMessageHandler ( ) is defined in the pe_template to register handlers with either the user interface or the model. Use this method as defined, changing the arguments for

adding handlers to supply the names of the messages and handlers you defined.

Registering Handlers for Updates to the User Interface

This code snippet from the Pe_template illustrates how to register the handler defined in the example in "Creating Handlers for Updates to the User Interface".

Note that you just use the code in the example as is. You don't need to specify anything other than the names of the messages and the objects that handle them. For each handler you register, add a call to the uiAdaptor. addHandler ( ) method. Change the arguments to the uiAdaptor.addHandler ( ) method to specify the names of the messages and handlers you defined. In this example, the name of the message is "sample_ui_message" and the name of its handler is SampleUIHandler ( ):

```
protected void createMessageHandlers ( ) throws
DuplicateHandlerException {
    uiAdaptor.addHandler ( "sample_ui_message",
        new SampleUIHandler (uiAdaptor, uiContainer) );
    //...
```

uiAdaptor and uiContainer are defined by the PresentationEngine class. Your handler needs to pass them as arguments so that its execute method has access to the user interface elements (GUI components).

Registering Handlers for Updates to the Model

If the message is to be handled in the Presentation Engine's data model, register the name of the handler in the model. This code is defined for you in the pe_template.

This code snippet illustrates how to register the handler defined in "Creating Handlers for Updates to the Model". Use the code in the example as is. Change the names of the messages and the methods that handle them:

```
protected void createMessageHandlers ( )
throws DuplicateHandlerException
{
    model. addHandler ("sample_model_message",
        new SampleModelHandler (model));
    //...
}
```

Handling Shutdown Events

ICE-T provides default shutdown handlers. The shutdown handlers are defined in the PresentationEngine class, not in pre_template. Developers who do not want to accept the default shutdown handling can write their own shutdown handlers. A new handler takes precedence over the previous handler for the same message name. To add a new handler, developers:

Write the handler

Use the same message name ("ICET_SHUTDOWN")

Use the methods provided for registering handlers in the Presentation Engine and the server program.

When a shutdown happens, the Communication Library notifies the application by means of the shutdown handlers. Shutdown handlers are identical to other message handlers, but have a preset message name ("ICET_SHUTDOWN"). To register a shutdown handler, the application code makes a call to addMessageHandler ( ) using the preset message name.

Registering Shutdown Handlers in the Presentation Engine

An application can register two shutdown handlers in the Presentation Engine, one with the UI Adaptor and one with the model. Registering shutdown handlers in the Presentation Engine is optional.

To register a shutdown handler on the uiAdaptor, use the following call:

uiAdaptor.addHandler ("ICET_SHUTDOWN", new shutdownHandlerUI (uiAdaptor, uiContainer));

To register a shutdown handler on the mode], use the following call:

model.addHandler ("ICET_SHUTDOWN", new shutdownHandlerModel (model));

Registering Shutdown Handlers in the Server Program

Your application can register one shutdown handler with the server program.

To register a shutdown handler in the server program:

SrvComm_addMessageHandler ("ICET_SHUTDOWN",
<functionPointerToShutdownHandler>);

Provide the name of the function that executes when the shutdown occurs,
(<functionPointerToShutdownHandler> in the example above).

Preparing the Server Program for Communication

To prepare the server program to communication with the client program, the developer "plugs in" the server program to ICE-T's Communication Layer. Connecting the server program to the server Communication Library, which is part of the ICE-T Communication Layer, is analogous to connecting the GUI to the Presentation Engine. In both cases, developers enable the exchange of messages, and these messages are handled locally by the components—the server program component and the Presentation Engine component.

The server program component in an ICE-T application must be written in any language that can call the C programming language. ICE-T provides both C and C++ language templates for handling messages and making calls to the server Communication Library. Use the appropriate template to start a new server program in C or C++, or add the template functions to an existing server program.

Note—If you have not done so already, copy one of the Templates subdirectories to the Application subdirectory. There are two subdirectories, one for C and one for C++. Each directory contains a server_template, one for C and the other for C++.

To enable communication between the server program and the client program:

Create message handlers in the server program that are analogous to the message handlers you created in the Presentation Engine. This step is described in "Handling Messages in the Server Program".

Make calls to the ICE-T server Communication Library.

The server program templates provide functions for message handling and communication. Developers just supply

the application-specific message names and their handlers. For C server programs use server-template. C. For C++ programs use server_template.cc. The templates are in the ICE-T installation directory under Templates/C and Templates/C++ respectively.

Each server program template calls a default main routine. The default main ( ) routines are provided for convenience. If you choose to write your own main ( ) routine, look at default_main.c or default_main.cc for guidance and follow the steps in "Modifying t3he Default main Routine (Optional)".

FIG. 28 is a table which describes the functions developers must fill in using the server program template. All three return integer values of 1 (TRUE) or 0 (FALSE). A return value of 1 (TRUE) indicates that the application startup can proceed. A return value of 0 (FALSE) indicates a problem that results in stopping the application. The remaining functions in the server program templates can be used as provided.

FIG. 29 is a table which illustrates Server Properties in accordance with a preferred embodiment.

Handling Messages in the Server Program

In developing a Presentation Engine, a key task is writing the code that handles messages. The same task is performed for the server program. It consists of similar steps:

Write the code that handles incoming events (typically requests for data)

Register the event handlers

To handle messages in the server program:

1. Write a function to handle each incoming message. Each function you write takes one argument, a SrvMessage( ) function, and has a void return type.

The function should extract the data from the message, and then call application logic to process it or send a message back to the client.

For example, here is a handler function in C that responds to a message with the data: employee name (empName) and employee number (empNumber):

```
void handleEmployeeRecord (SrvMessage *message) {
    SrvData *empName;
    SrvData *empNumber;
    char *name;.
    int num;
/*
* disassemble incoming data
*/
empName= SrvMessage_getDataElement (message, 0 );
empNumber=SrvMessage_getDataElement (message, 1 );
name=SrvString_getValue (empName);
num =SrvInteger_getValue(empNumber);
/*
* Now process the data..
*/
lookupEmployee (name, number);
Here is the handler function in C++:
void handleEmployeeRecord (SrvMessage *message) {
/*
* disassemble incoming data
*/
SrvData *empName= message.>getDataElement (0);
SrvData *empNumber=message >getDataElement (1);
char *name=empName >getValue ( );
int num =empNumber->getValue( );
/*
* Now process the data...
*/
lookupEmployee (name, number);
```

2. Resister the message handlers.

Fill in the createMessageHandler ( ) function in the server-template to register the handlers with the server Communication Library.

Note that you just use the code in the example as is. You don't need to specify anything other than the names of the messages and the functions that handle them. For each handler you register, add a SrvComm_addMessageHandler ( ) call. The arguments to the functions in this example are the application-specific names of the messages and handlers you defined in Step 1.

```
int createMessageHandlers ( ) {
SrvComm_addMessageHandler ("Double", handleDouble);
SrvComm_addMessageHandler
("firstNameEntered", handleSearchMessage);
//...
return 1;
}
```

Modifying the Default Main Routine (Optional)

Once the server component of the application includes definitions for how it handles messages and registers those handlers, it can call the server Communication Library. Those calls are defined in the default main routines provided by ICE-T: default_main. c and default_main. cc.

default_main. c and default_main. cc are in the ICE-T installation directory under Server/Main.

This section describes the order and usage of the calls to the server Communication Library.

If you create your own main or modify the default, follow the instructions in this section for including the calls that enable the server program to work with the ICE-T Communication Layer.

To connect the server program to the Communication Library:

1. Include the ICE-T header files.

The default_main files include the header files already:

```
#include    "SrvData.h"
#include    "SrvComm.h"
#include    "SrvMessage.h"
```

2. Call create_SrvComm ("tcp") to create the Communication Library's data structures.

```
char *protocol="tcp";
create_SrvComm(protocol);
```

3. Call setSrvCommProperties ( ). (Optional) Set server properties. The setSrvCommproperties ( ) function is typically implemented in a separate file that is linked to the server program as part of the build process.

4. Call createMessageHandlers ( ).

See "Handling Messages in the Server Program" for information about how to use createMessageHandlers ( ).

5. Call SrvComm_createSocket ( ) to create the listener socket and output the port number.

Note—Do not write anything to stdout before you make the SrvComm_createSocket ( ) call.

6. Initialize the application.

7. Call SrvComm_acceptClientConnection ( ) to a ccept the connection from the client.

Note that the order of Step 7 and Step 8 may be interchanged.

8. Finally, call SrvComm-start ( ) to start the Communication Library's loops.

This routine does not return until everything is finished.

Building ICE-T Applications

Building an ICE-T application consists of modifying a make file for the client and server programs, and then using it to make both programs.

ICE-T provides a makefile (Example.mk) for both the Presentation Engine and the server programs. Example.mk is in the ICE-T installation directory under Templates/C or Templates/C++. These files are copied to the/Applications directory.

To use the makefile, modify it to specify the:

Compiler location

Libraries location

ICE-T installation directory (optional)

Client and server source file locations and names

1. Open Example.mk in an editor and follow the instructions in the file.

2. Supply the locations of the compiler, libraries. If you have moved the ICE-T installation directory, supply its new location. There is no need to change the location of the ICE-T installation directory. If you choose to move it, supply the name of the location in the TCET-INSTALL-DIR macro.

The macros for which you provide values are shown below as indicated in the code:

```
############################
### Environment Configuration ###
############################
#
# Change the following lines to indicate the location of your compiler.
#
COMPILER_BIN=
COMPILER_LIB=
#
# Change the following line to indicate the location of the ICET
installation
# directory (either absolute or relative to the current directory).
# The default value (../..) allows you to build an example in-situ in the
# installation hierarchy.
#
ICET - INSTALL - DIR=../..
```

3. Change the macros for the Presentation Engine source files. This example specifies the Java files for the Presentation Engine template (pe_template.java) and a user interface file named myGui.java. The macros for which you provide values are shown here in bold type. Change the names of the files to those used in your application:

```
############################
###     PE      ###
```

-continued

```
### (Configurable Macros)###
############################
#
# Change the following macro to add your java files
#
PE_SOURCE5. java= \
    myGui.java \
    pe_template.java \
    #end
#
# Change the following macro to indicate which of the java classes
# is the top level class (i.e. the subclass of presentationEngine).
# Note: specify this class without any extension, e.g.:
# PE_MAIN_CLASS=myPresentationEngine
#
PE_MAIN_CLASS=pe_template
```

Example.mk specifies files for the server program template (server_template). The macros for which you provide values are shown here in bold type. There is a macro for C source files and one for C++ source files. Change the names of the files to those used by the server program in your application:

```
############################
###     Server      ###
###  (Configurable Macros)   ###
############################
#
# change the following macro to add .cc (C Plus Plus) files
#
SERVER_SOURCES.cc=\
    #end
#
# change the following macro to add .c (C) files
#
SERVER_SOURCES.c=\
    default_main.c \
    server_template. c \
    #end
#
# change the following macro to indicate the name of the
# server executable
SERVER = ServerTemplate
```

Build the client program with the following command:

% make -f Example.mk pe

Build the server program:

% make -f Example.mk server

Both pe and server are defined for you in the Makefile.

Testing ICE-T Applications

You can test the Presentation Engine alone or with the server program. You have to build the programs before you test them.

Run the Presentation Engine class by itself using the java command.

For example:

% java myPresentationEngine

To test the Presentation Engine with the server program:

1. After you have made the server program, run it to get the port id.

This example shows the result of running a server program named myServer:

% myServer

//ICE LOG messages deleted from this example.

37526 129.146.79.147

ICE: :ApplicationServer: port=37526 ServerIP= 129.146.79.147

2. Supply the server machine IP and the port id as arguments to the java command. For example, using the machine and port id from Step 1:

% java myPresentationEngine 129.146.79.147 37526

3. To test the Presentation Engine in debug mode, use the -debug option to the java command. The—debug option should be the first argument after the class name:

% java myPresentationEngine—debug 129. 146.79. 147 37526

Configuring and Deploying ICE-T Applications

In addition to tools for developing a Presentation Engine, ICE-T provides these components to support application deployment, execution, and administration:

Installation scripts

The following installation scripts are in the ICE-T installation directory under/bin:

ice-httpd-setup

Sets up the directories you need on the server.

ice-install-access

Installs the Access Layer on the server.

ice-app-install

Installs each compiled Presentation Engine and server program on the server.

Access Layer

Acts as gatekeeper for HTTP communication between the client and server nodes. The Access Layer is a cgi-bin executable that identifies and authenticate users. You can modify the e files used by the Access Layer and then use the supplied makefile (Access.mk) to build a customized Access program for use with ICE-T server applications.

Application startup applet template (Java)

A template for making Java applets that launch ICE-T applications. The template is in the ICE-T installation directory under StartApplet.

Web server (user must install)

Supports HTTP connections; enables Web access and the use of a browser front end for executing an ICE-T application.

Note—Sun internal users can install the Web server from /home/internet/CERN/httpd/. The directory contains a README file with installation instructions.

Deploying and maintaining ICE-T applications involves these steps:

1. Using a stamp applet and HTML pages to launch ICE-T applications
2. Setting up the Web server
3. Customizing (optional) and installing the Access Layer
4. Running installation scripts for Presentation Engines and server programs
5. Configuring application management files

Using Startup Applets and HTML Files

Compiled Presentation Engines can run as applets in a Java-enabled browser. To enable users to launch ICE-T applications from a browser use the named ICE-T templates to:

Create a startup applet for each application. Use the startAppletDevIR. java template.

"Using the Startup Applet" describes this step.

Create a top-level HTML file with links to each application. This file serves as a "splash page" identifying the applications available and including links to an HTML file for each application. Use splashTemplate.html.

"Creating a Top-Level HTML File" describes this step.

Create an HTML file for each application. Use appTemplate.html.

"Creating Individual Application HTML Files" describes how.

Using the Startup Applet

A startup applet provides a way to launch an ICE-T application from a Web page. The startup applet:

Starts the HTTP access to the server

Authenticates the user

Starts the server program

Downloads the Presentation Engine as an applet to the Web page or opens the applet in a separate user interface window (developer's choice)

You can use the startAppletDevIR.java template to launch the applications you build, or create your own applet. The template is completely generalized so that it can be used for any of your applications. You supply the application name in a separate parameter to the applet tag. (See "Creating Individual Application HTML Files".)

A complete code listing for startAppletDevIR.java is in Appendix B.

By default, the startup applet opens the Presentation Engine in the same window. To have the Presentation Engine appear in a new browser window, open startAp~letDevIR.java and follow the instruction in the file:

```
AppletContext cont = app.qetAppletContext( );
    if (cont != null) {
        System.err.println ("Showing document: ["+newURL+"]");
        cont.showDocument (newURL);
        /*
        * To show the applet in another window, replace the
        * the line above with the commented line below.
        */
        //cont.showDocument (newURL), "new_window");
}
```

Creating a Top-Level HTML File

You need a a top-level HTML file, or "splash page," to present the list of available applications with links to the application-level HTML pages for them. When a user chooses a link, the link takes them to an application-level HTML page.

ICE-T provides a default HTML file for the "splash page." The file, called spllashTemplate.html, is in the ICE-T installation directory under StartApplet. You can use the default file or make one of your own.

To create a top-level Web page for listing the links to your application Web pages:

1. Copy splashTemplate.html to another file. For example:

% cp splashTemplate.html myAppSplashPage.html

2. Open the file in an editor.

3. Provide a title for the page and any text you want to include about the application(s) listed there.

4. Supply the name and the URL of the application-level HTML page for each listed application. For example, if you used appTememplate.html to create an HTML file for an application named MyApplication1:

```
<html>
•
•
•
<a href="MyApplication1.html">
MyApplication1
</a>
•
•
•
</html>
```

5. Copy the file to the appropriate location on your Web server.

The result is a Web page with a link to the named application:

MyApplication 1

When a user chooses this link, the browser loads an application-level HTML page with the startup applet for the application.

Creating Individual Application HTML Files

Think of the application-level HTML file as the Presentation Engine startup page. This page contains the startup applet that results in the Presentation Engine being downloaded.

When the user launches the stamp applet, the startup applet takes the name of the application from the applet parameter, the user name, and the password and sends these using HTTP to the Access Layer on the server.

To create an application-level Web page:

1. Copy alDIDTemlDlate.html to another file. For example:

% cp appTemplate.html myAppPage.html

2. Open the file in an editor.

3. Include instructions to the user on how to launch the application. (Optional) startAppletDevIR.java defines a Send button and a class (sendBtn) that handles the user's input. appTemplate.html includes default instructions for using Send. If you want to change the user's interaction with the stamp applet, you would need to change the sendBtn class and the instructions in appTemplate.html.

4. Specify the name of the startup applet. If you have copied startAiDiDletDevIR.java, given it another name, and compiled it, supply that name to the applet tag:

<applet code="startAppletDevIR.class" width=400 heicJht=400>

5. Provide the application name as a parameter to the applet tag. For example:

<param name=AppName value="MyApplication1">

6. Provide the name of the access program to use for this application. The default value is "Access." You may have customized the default access file and given it another name. If so, provide the name as a value to the Access parameter.

<param name=Access value="Access">

Be sure that the file you create from appleTemplate.html contains the end applet tag </applet>.

5. Here are the tags for a minimal HTIVIL file using startAlppletDevIR, an application named "MyApplication1", and the default access program name:

<html>

<blockquote>

Please provide Username and Password and press the "Send" button to launch the application.

</blockquote>

<hr>

<applet code="startAppletDevIR.class" width=400 height=400>

<param name=AppName value="MyApplication1">

<param name=Access value="Access">

</applet>

</html>

When the user launches the startup applet and sends the user data and the application name to the server, ICE-T's Access Layer:

Authenticates the user name and password

Downloads the Presentation Engine as an applet in a Web page

As part of the process of determining access and downloading Presentation Engines, the Access Layer relies on an application configuration file to provide information about the location and names of program components. ICE-T's Access Layer installation script generates the application configuration file automatically. That configuration is the basis for generating an HTML wrapper file in which to download the Presentation Engine. You can accept the defaults and let your application use the generated HTML wrapper, or you can customize the application configuration file so that it generates a customized HTML file to hold the Presentation Engine. See "Configuring Applications" for more information.

Setting up the Web Server

Note—Complete the following Web server setup before installing the ICE-T application, that is, before running ice-app-install.

This step establishes a Web server file structure for the ICE-T files that manage access to and delivery of client applications. To establish this file structure, run the script provided in the ICE-T installation directory under /bin/ice-httpd-setup.

Before you run this server setup script, you should establish links to the cgi-b in directory and the httpd-docs directory.

Create a symbolic link to the cgi-bin and httlDd-docs directories using the UNIX ln command.

Become superuser

Make the links:

% in -s <absolute path to cgi-bin>/cgi-bin

% in -s <absolute path to httpd-docs>/WWW-docs

If you cannot create these links, provide the absolute path names to cgi-bin and httpd-docs as arguments to the ice-httpd-setup script. If you have not created the links, you will need to change the Access Layer's default access properties file to specify the cgi-bin and httpd-docs locations. (See "Customizing the Access Layer"). If you have created the links, run ice-httpd-setup without arguments.

From the ICE-T installation directory, run ice-httpd-setup.ice-httpd-setup takes these arguments:

-cqi-bin —takes the location of the c-i-bin directory. The default is/cgi-bin.

-httpd-docs—takes the location of the httpd/docs directory. The default is/www-docs.

Note—Run ice-htttpd-setup once for each machine that is an ICE-T Web server.

For example:

% cd <ICE-T installation directory>

% /bin/ice-httpd-setup

ice-httpd-setup establishes a Web server file structure for the files that manage access to and delivery of requested client applications.

ice-httpd-setup performs these tasks:

Creates an ICE directory under the -cgi-bin directory

Creates a link to cgi-bin/ICE from the httpd-docs directory

Creates an application startup directory under the ICE directory

Creates an ICEAppRepository directory

ice-httpd-setup takes additional optional arguments. Run the setup file with the -help quailifer to get a list of the arguments and their defaults:

% cd <ICE-T installation directory>

% /bin/ice-httpd-setup-help

Customizing the Access Layer

This step is optional. The Access Layer uses authentication and access properties files to manage access to ICE-T applications. Developers can use the files provided by ICE-T or modify them to specify their own access requirements or to create a number of access subsystems for use with different applications.

The Access Layer uses:

default_authentication.cc

default_authentication.cc contains a single default authentication function that checks the application name, user name, and password. By default, all users are authenticated. Modify the tile to specify how to authenticate users.

default_access_properties.cc

default_access_properties contains several functions for controlling access log tile properties, as well as functions for specifying the Access directory, and password scrambling keys.

With the exception of the cqi_bin_location property, (See table below) no modification is necessary for these tiles, but you may choose to change the defaults for authentication and access properties. The files are in the Access subdirectory in the ICE-T installation directory.

The table appearing below describes properties that can be modified in default_access_properties.cc:

Table Default Access Properties

Property	Default	Description
cgi_bin_location	/cgi -bin	If you have not created a link to cgi-bin (as described in "Setting up the Web Server", provide a cgi -bin location here.
log_destination	logfile	The return value determines where log entries are written. Accepted values: console, logfile. If you choose to write the log to a file, set the file location and name using the logfile property.
logging_level	all	Determines to what level the Access Layer logs

-continued

Table Default Access Properties

Property	Default	Description
		access attempts. Accepted value: none, all, starts, errors, debug.
logfile	<cgi_bin_location>/ ICE/Access logs/ Access - pid.log	
scrambler_key	Uses the scrambler key provided by ICE-T.	Return value is used as a key to unscramble the user name and password sent when the user launches an application. If you change the scrambler key, change the scrambler key in the startup applet also.
sessionEnvironment		Sets up the calling environment for the server program. You can make calls to putenv ( ) to set environment variables whose values can be accessed with getenv ( ) from the server program.

To implement the authenticate ( ) function in default_uthentication.cc:

1. copy the file from the ICE-T installation's Access directory.

2. Follow the instructions in the file to implement the authenticate ( ) function.

3. In Access.mk, replace this file name with yours.

To change the defaults in default_access._properties.cc:

1. Copy the file from the ICE-T installation's Access directory and give it a name that suits the application.

2. Modify the file to supply application-specific values.

3. Edit Access.mk to replace the default file names with the files you edited. Access.mk is in the Access directory. The user-modifiable section is marked.

To substitute the new file name for default_authentication.cc, change the second line:

AUTHENTICATION_SOURCES.cc=\

default_authentication.cc \

To substitute the new file name for default_access_properties.cc, change the second line:

ACCESS_PROPERTIES_SOURCES.cc=\

default_access_properties.cc \

4. Run Access.mk.

% make -f Access.mk

5. After you configure the Access Layer, run ice-install-access. By default, Access.ink creates a an executable named CustomAccess. If you have changed the default name of the access program, use the—source option to ice-install-access and specify the name you have given the access program. For example:

% ice-install-access-source myAccess

Installing the Access Program

The ICE-T Access Program installation script:

Installs the Access program on the server in the cgi-bin directory that you specify

Installs the startup applets in the/www-docs/ICE/start directory.

To install the Access program, run ice-install-access from the ICE-T installation directory:

% cd <ICE-T installation directory>

%/bin/ice-install-access

ice-install-access takes four optional arguments:

-cgi-bin <cgi-bin location>

The defau/t is/cgi-bin.

-source <Access executable name>

The name of the Access program that you are copying from. The default is Access.

-dest <Access executable name>

The name of the Access program that you are copying to. The default is Access. Change the destination if you want to use different access programs for different applications.

-help

Displays the arguments and their defaults.

Installing the ICE-T Application

ICE-T provides a script for installing applications. ice-app-install performs these tasks:

Installs the client and server executable files for the application

Creates an application directory

Installs the server libraries

Installs the Java packages used by the Java Presentation Engine class

Creates an application configuration file (appConfigFile), or installs a specified one

If you already have an appconfigFile to which you have made changes, you can avoid having the application installation overwrite it. Use the -appConfigFile <file>option.

Copies the alDlDConf igFile to the application directory

Installing the ICE-TApplication with ice-app-instal1

Note—Run ice-app-install for each application you want to install.

From the ICE-T installation directory, run ice-app-install with the required arguments:

-appName—the name of the application

-peClass—the name of the Presentation Engine class

-peSource—the location of the Presentation Engine you want to install

-serverName—the name of the server program

-serverSource—the location of the server program you want to install

If peSource and serverSource are the same directory, you only need to specify one of them.

ice-app-install takes additional optional arguments: ice-app-install is in the /bin directory of the ICE-T application installation directory. You can run the file with the -help option to get a list of the arguments and their defaults:

% cd <ICE-T installation directory>

%/bin/ice-app-install-help

Installing the ICE-T Application with the Provided Makefile

An alternative to running ice-app-install, is to use the Example.mk Makefile to install the completed application. ExamiDle.mk has defined a target for installing the application on your Web server. If you have set up the Web server, creating the symbolic links as described in "Setting up the Web Server" then you can use ExamiDle.mk with the following argument:

% make -f Example.mk app-instal1

If you did not create a symbolic link to the cgi-bin directory as described in "Setting up the Web Server", specify the cgi-bin location in the Example.mk file (use the CGI_BIN_LOCATION macro and then run make on the aIDiD—install target:

% make -f Example.ink app-install

Configuring Applications

The ICE-T application installation script (ice-app-install) generates a default application configuration file (appConfigFile). When a user starts an application by launching a startup applet, the Access Layer uses the Application Manager to look up values for server and client program locations and names in appConfigFile. Using the configuration file, the Application Manager generates an HTML wrapper for presenting Presentation Engines as applets in a Web browser execution environment. (See "Using Startup Applets and HTML Files" for more information about how to use startup applets for ICE-T applications.)

To complete the last step in deploying an application to a Web Browser you use one of two ways to supply application-specific values in the HTML wrapper:

Run ice-app-install with the required arguments as described in "Installing the ICE-T Application" and let it create the default aiDiDConfigFile.

Or, create your own application configuration file by modifying the automatically generated appConfigFile.

By default, ice-app-install creates the application file in the /cgi-bin/ICE/ICEAppRepository directory and names it <appName>.appconf.

To customize the appConfigFile generated by ice-app-install:

1. Open the generated configuration file (<appName>.appconf) in an editor.

2. Supply the names for each application in the appropriate tags in each appConfigFile. You are required to supply application information in these tags:

<peClass>—the name of the Presentation Engine class

<serverName>—the name of the server program

<PresentationEngine>—specifies where the Presentation Engine appears on the Web page.

The Application Manager replaces the <Presentation Engine> tag with a Java <applet> tag that specifies the named Presentation Engine (peClass).

3. Supply messages to return to the browser if user authentication or application stamp fails. The template contains tags for authentication failure and application startup failure messages.

The appConfigFile contains optional tags for you to specify a Web page title, headings, and text, just as you would for any Web page.

The Application Manager also uses a properties file to set what applications are returned to the client. By default, the Application Manager returns the application name passed to it by the Access Layer. You can specify another application to return by modifying default_appmgr_properties. cc.default_appmgr_properties.cc is described in "Customizing the Access Layer".

Presentation Engine Template

ICE-T provides a null application that you can use as a template for Presentation Engines. You can find the file for the template in the ICE-T application installation directory under /Templates/pe_template.java.

Code Example A-1 pe_template.java Listing

```
import sunsoft.ice.pe.*;
import java.net.*;
import java.io.*;
import java.applet.*;
import java.util.*;
import java.awt.*;
/*
* This is a sample template of the ICE-T Presentation Engine - it needs to
* be filled with the actual ui, names of messages and handlers to create
* a working PE.
*/
// Extend the PresentationEngine class
public class pe_template extends PresentationEngine {
// Constructor
public pe_template( ) {
  // This is required. It sets up the internal PE components
  super( );
}
```

Code Example A-1 pe_template.java Listing (Continued)

```
/**
* This is called when the current applet terminates.
* It needs to be customized for handling the termination
* of the current applet.
*/
public void terminate (String reason) {
  super. PEterminate(reason);
}
/**
* This can be used to do any local initializations that may be required in
* the PE. It is called before all the connections are setup with the server
* so no messages should be sent from this function.
*/
protected void initializeApplication( ) { }
/**
* createUI creates the UI components of the applet.
* It uses the "ui" object of "gui" class which is generated
* by the SpecJava gui builder.
*/
/**
* Specify the UI of the application in this function.
*/
protected void createUI( ) {
  // create local awt components to map the ones generated
  // by specJava. For example:
}
/*
* createModel Creates the data items in the Model. Some data items
* can be observable. These items which have an associated observer
* function that gets called when the observable is updated.
*/
// This function is optional. It is only needed if the
// application wants to do some client-side application logic
// Uncomment the lines below if you plan to do client side processing
// in the Model.
```

Code Example A-1 pe_template.java Listing (Continued)

```
protected void createModel( ) {
//create the observable object e.g. PeHashtable, PeVector, PeString . . . )
//Attach observer to the observable.
//put the observables into Model
}
/**
* provide the mapping of Message names to handlers for the inbound
* Messages to the PE from the UI or Comm adaptors.
* The Handlers for the UI and Model messages should
* be provided in this function.
*/
// This function is required. It keeps the mapping of UI events
// registered with the uiAdaptor and the model events registered
// with the model
protected void createMessageHandlers( ) throws
DuplicateHandlerException {
// UI maps
uiAdaptor.addHandler ("sample_ui_message",
new SampleUIHandler (uiAdaptor, uiContainer));
//. . .
// Model Maps
model.addHandler ("sample model_message",
new SampleModelHandler(model));
```

-continued

```
//. . .
}
/*
* main is used when running as a standalone java application,
* (i.e. not in a browser environment), and is intended to do
* all the things that the browser would do for the applet.
```

Code Example A-1 pe_template.java Listing (Continued)

```
*
*/
public static void main(String args[ ]) {
pe_template pe = new pe_template( );
pe.isApplet = false;
pe. getCommandLineArgs (args);
pe.init( ); pe. start ( );
}
}
////////////////////// UI Message Handlers
////////////////////////////
class SampleUIHandler extends PeUIHandler {
public SampleUIHandler( PeUIAdaptor adaptor, PeUI uiContainer) {
super (adaptor, uiContainer);
}
public boolean execute (Object caller, PeMessage message) {
// decode the record send by the server
// update the ui
return true;
}
}
////////////////////// Model Message Handlers
////////////////////////////
class SampleModelHandler extends PeModelHandler {
public SampleModelHandler ( PeModel model ) {
super (model);
}
public boolean execute (Object caller, PeMessage message) {
```

Code ExampleA-1 pe_template.java Listing (Continued)

```
// decode the record send by the server
//update the model
return true;
}
}
////////////////////// Model Observers
////////////////////////////
class SampleObserver extends PeObserver {
public SampleObserver ( PeModel model) {
super (model);
}
public void update (Observable o, Object arg) {
}
}
```

Startup Applet Template

startAfalaletDevIR. java is a Java applet that launches ICE-T applications. The file is generalized to run any ICE-T application. You don't need to change it unless you want to change the user's interaction with the applet. The applet currently asks for username and password and provides a Send button to launch (start up) the application. The file is in the ICE-T installation directory under/StartApplet. For instructions on how to use this file, see "Using Startup Applets and HTML Files".

Code Example B-1 startAppletDevIR.java Listing

```
/**
* This is a sample applet that can be used for gathering specific
* user information before displaying the ICE-T application. It can
* be customized as needed. It is loaded into the browser using the
* index.html file provided.
*/
import java.net.*;
import java.io. *;
import java.applet. *;
import java.util. *;
import java.awt. *;
import sunsoft. ice.ui. *;
public class startAppletDevIR extends Applet {
Code ExampleB-1 startAppletDevIR.java Listing (Continued)
public startAppletDevIR ( ) {
}
public void init ( ) {
setLayout (new ColumnLayout ( ));
Panel fnamep = new Panel ( );
fnamep.setLayout (new FlowLayout (FlowLayout. LEFT));
fnamep.add (new Label ("Username:"));
TextField fnameText= new TextField (" ",20);
fnamep.add(fnameText);
add(fnamep);
Panel passwdp = new Panel ( );
passwdp.setLayout (new FlowLayout (FlowLayout.LEFT));
passwdp.add (new Label("Password: "));
TextField passwdText= new TextField(" ",20);
passwdText.setEchoCharacter('.');
passwdp.add (passwdText);
add(passwdp);
Panel btnp = new Panel ( );
btnp.setLayout (new FlowLayout ( ));
AButton cbtn = new Abutton ("Send");
cbtn.setAction (new sendBtn ( ));
cbtn.addClientData ("applet",this);
cbtn.addClientData ("username",fnameText);
cbtn.addClientData ("password",passwdText);
btnp.add (cbtn);
add(btnp);
show ( );
}
public String getAppName( ) {
appname = getParameter ("AppName");
return appname;
}
public String getAccessName ( ) {
String acc = getParameter ("Access");
Code ExampleB-1 startAppletDevIR.java Listing (Continued)
if (acc == null 11 acc.equals(" ") ) acc="Access";
return acc;
}
public String getAppHost ( ) {
System.err.println ("Applet: "+isApplet);
if (isApplet) {
URL docBase=getDocumentBase ( );
System.err.println("Document base: "+docBase);
apphost=docBase.getHost ( );
if (apphost==null) apphost=" ";
int port=docBase.getPort ( );
if (port != -1) apphost = apphost + ":"+port;
System. err.println("Server is: {"+apphost+"}");
} else {
apphost=" ";
}
return apphost;
}
public static void main (String agrs[ ]) {
Frame f = new Frame ("ICE-T startup applet");
startAppletDevIR tap = new startAppletDevIR ( );
tap.isApplet=false;
tap.init ( ); f.add ("Center", tap);
f.resize (500,400);
f.show ( );
tap.start ( );
}
public boolean isApplet=true;
private String appname = new String(" ");
```

-continued

```
private String apphost = new String(" ");
}
/*
* sendBtn handles the "Send" button activation. The execute
* member of this class is called when the "Send" button is
* pressed. It collects all the relevant user information and
Code ExampleB-1 startAppletDevIR.java Listing (Continued)
* sends it to the Server via HTTP. This is done by executing
* the Access function in the server's cgi-bin.
* [Note: the username and password is scrambled so it will not
*       be displayed in the URL field of the browser.]
*/
class sendBtn extends Activity {
public boolean execute (Object caller, Event event) {
AButton btn= (AButton) caller;
startAppletDevIR app = (startAppletDevIR) (btn.getClientData ("applet"));
TextField name = (TextField) (btn.getClientData
("username"));
TextField passwd = (TextField) (btn.getClientData
("password"));
String servername = app.getAppHost ( );
URL newURL = null;
String username=name.getText ( );
String userpasswd=passwd.getText ( );
String appname = app.getAppName ( );
String accessName=app. getAccessName ( );
AccessScrambler scrambler=new AccessScrambler ( );
System.err.println ("Scrambling ("+username+", "+
//userpasswd+", "+
appname+")");
String scrambled=scrambler.scrambleUser (username, userpasswd, appname);
System.err.println(" scrambled ==> ["+scrambled+"]");
String uri ="http://" + servername +
"/cgi-bin/"+accessName + "?" + appname +
"+" + scrambled;
System.err.println (url);
try {
newURL = new URL(url);
}
catch (Exception e) {
System.err.println ("Exception Getting newURL from ("+url+"): "+e);
}
AppletContext cont = app. getAppletContext( );
if (cont != null) {
Code ExampleB-1 startAppletDevIR.java Listing (Continued)
/*
* The call to ShowDocument makes the httpd request to the Access
* cgi-bin executable.
*
* Call showDocument with one argument (the URL) to show the
* PE in a single window; call with two arguments (the URL "new_window")
* to show the PE in a new browser window.
*
* We're currently defaulting the call to bring up a new browser
* window because of the way browsers handle backing out of html
* pages with java applets - - if you go too far back, then the applet
* is destroyed. We put the PE in a new window to minimize the chance
* of going too far back simply because you want your browser back,
* but don't intend to kill your PE.
*
*/
System. err.println ("Showing document in new window: ["+newURL+"] in
window: ("+appname+")");
cont.showDocument (newURL, appname);
//cont.showDocument (newURL);
//System.err.println ("Showing document: ["+newURL+"]");
}
return true;
}
}
```

Server Program Templates

This appendix contains code listings for the following templates:

server_template.c

default_main.c

server_template.cc

default_main.cc

Chapter 2, "Building Program Components describes the location and use of these templates. See "Handling Messages in the Server Program" and "Modifying the Default main Routine (Optional)" for more information.

C++ Files

ICE-T provides a serter program template and a default main routine for application developers using C++ to develop server programs.

C++ Server Program Template

Code Example C-1 server_template.cc Listing

```
/*
* server_template.cc
*
* This is a template of a ICE-T server-application in C++
*/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
// these are required to get the server side ICE-T components
#include "SrvData.hh"
#include "SrvComm.h"
#include "SrvMessage.hh"
#define false 0
#define true 1
/***********************************
* The Message Handler functions *
***********************************/
/* This is the Message handler function that gets called when a specified
* message is received from the client. It has only one argument
* (msg) that contains the message name and message data.
*/
void handleMessage (SrvMessage *msg) {
/*
* disassemble incoming data
*/
```

Code Example C-1 server_template.cc Listing (Continued)

```
/*
SrvData *data;
data=message->getDataElement (0);
*/
/*
* The application "Logic" */
*/
/*
* Assemble outgoing data
*/
// send a reply if necessary. <replyData> can be of the types
// provided in SrvData.hh
/*
SrvMessage *reply=new SrvMessage ("replyMessageName");
reply->addlDataElement (<replyMessageData>);
SrvComm_sendMessage (reply);
*/
}
//This is an example of a shutdown handler on the server
void shutdownHandler (SrvMessage *shutdownMsg) {
char *shutdownType= ((SrvString *) shutdownMsg->getDataElement (0))-
>getValue ();
char *reasonString= ((SrvString *) shutdownMsg->getDataElement (1))->getValue
( );
if (!strcmp (shutdownType,"ICET_CLIENT_SHUTDOWN")) {
fprintf (stderr,"shutdownHandler: Detected CLIENT SHUTDOWN event.
Reason=%s\n", reasonString);
} else if (!strcmp (shutdownType,"ICET_CLIENT_DISCONNECT")) {
fprintf (stderr,"shutdownHandler: Detected CLIENT DISCONNECT event.
Reason=%s\n", reasonString);
} else if (!strcmp(shutdownType,"ICET_SERVER_SHUTDOWN")) {
fprintf (stderr,"shutdownHandler: Detected SERVER SHUTDOWN event.
Reason=%s\n", reasonString);
} else if (!strcmp (shutdownType,"ICET_INTERNAL_SHUTDOWN")) {
```

Code Example C-1 server_template.cc Listing (Continued)

```
fprintf (stderr,"shutdownHandler: Detected INTERNAL SHUTDOWN event.
Reason=%s\n", reasonString);
} else {
fprintf (stderr,"Shutdown Handler: unknown msg type (%s). Reason=%s\n",
shutdownType, reasonString);
```

-continued

```
}
}
/*****************************************************************
*
* Functions required by default-main, that set the                *
* SrvComm library properties (setSrvCommProperties) and           *
* initialize the applicationService (InitializeApplicationService)*
*                                                                 *
*****************************************************************/
int setSrvCommProperties ( ) {
/* The server timeout is how long the server waits between receiving
* messages from the PE before it initiates the tim,out shutdown procedure.
*/
SrvComm_setServerTimeout (300); /* in seconds */
/* The accept timeout is how long the server will wait to accept a connection
* from the client before it shuts down.
*/
SrVComm_setAcceptTimeout (400); /* in seconds */
return 1;
}
int createMessageHandlers ( ) {
SrvComm_addMessageHandler ("SampleIncomingMessage", handleMessage);
SrvComm_addMessageHandler ("ICET_SHUTDOWN", shutdownHandler);
return 1; // (return 0; if there is a problem here)
}
int initializeApplication ( ) {
return 1; // (return 0; if there is a problem initializing the application)
}
Default main for C++
Code Example C-2 default_main.cc Listing
/*****************************************************************
* default_main.cc
*
* This file implements the default main ( ) used
* by an ICE-T server executable.
*
* The structure of this main is that it calls
* all the functions necessary to run
* the ICE-T SrvCom library, and provides several
* places for the application writer to include
* application-specific code. These functions
* are implemented by the application-writer,
* and are usually implemented in a separate file, that gets
* linked in as part of the build process.
*
* These functions all return int values, which are used to determine if
* the startup of the application should continue or should be aborted.
* In this way the application code has a hook for indicating when something
* goes wrong (can't connect to database, or something), and passing this
* information on to the application so startup doesnt continue.
* Return 1 (TRUE) to indicate that everything is OK, or return 0 (FALSE)
* to indicate a fatal problem that should result in aborting the application
startup.
*
* int setSrvComProperties ( )
* Fill in this function to set properties on the SrvComm library.
* This function is called after the SrvCom library is created, but before it accepts
* a connection from the client.
*
* int createMessagHandlers ( )
* Fill in this function to register the message handlers.
* This function is called imediately after setSrvCommProperties ( ).
*
* int initializeApplication ( )
* Fill in this function to start up the application-specific code.
* It is called after the connection is accepted from the client,
* and before the message reader and handler loops are started.
*
* Code Example C-2 default_main.cc Listing (Continued)
*
* NOTES:
* This main ( ) routine is provided for your convenience,
* but you are not required to use it. You are
* free to write your own main ( ) routine, provided
* you follow the requirements for ordering the
* calls to the SrvComm library:
* 1.        First call create-SrvComm("tcp"); to create
*           the Comm library's data structures.
* 2.        Set the properties on the Comm library and
```

-continued

```
*           register message handlers and other handlers.
* 3.        Call SrvComm_createSocket ( ) to create the
*           listener socket and output the port number
*           (Note: do NOT write anything to stdout before
*           making this call).
* 4.        Initialize the application.
* 5.        Call SrvComm_acceptClientConnection ( ) to
*           accept the connectio fromthe client.
*           (Note: steps 4 and 5 may be interchanged)
* 6.        Finally call SrvComm_start ( ) to start the
*           Comm library's loops. This routine does not
*           return until everything is finished.
*
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include "SrvComm.h"
extern int setSrvCommProperties ( );
extern int createMessageHandlers ( );
extern int initializeApplication ( );
/*****************************************************************
*       The default Main                                         *
*                                                                *
* ICE-T provides a default main( ) routine                       *
* for ICE-T server applications.                                 *
*                                                                *
*****************************************************************/
Code Example C-2 default_main.cc Listing (Continued)
int
main(
intargc,
char**argv,
char**envp
)
{
char *protocol="tcp";
create_SrvComm(protocol);
/*
* Call user-defined function to set the properties on the Comm Library.
* Then call user-defined function to create the Message handlers
* and add them to the Comm library.
*
* The setSrvCommProperties function is where Comm Library properties
* are set. All properties have reasonable defaults, so none of them are
* required to be set.
*
* The createMessageHandlers function is where the message handlers are
* added, and other handlers (shutdown handler, default message handler,
* etc) are registered.
*
* Requirements on these two functions:
* 1. These functions may NOT write anything to stdout, since the Access
* layer expects the first thing on stdout to be the port number that
* the server will be listening for a client connection on.
*
*/
if (!setSrvCommProperties ( ) ) {
exit (0);
}
if (!createMessageHandlers ( ) ) {
exit(O);
}
/*
* Create the socket that the client will connect to.
* This routine also is responsible for generating (and printing to
Code Example C-2 default_main.cc Listing (Continued)
* stdout) the port number so the Access layer can generate the
* html code that passes the port number to the client applet
* (presentation engine).
*
* (NOTE: Do not write anything to stdout before this routine is called.
*       Writing to stderr is OK, however.)
*/
if (SrvComm_createSocket ( ) != SRVCOMM_STATUS-OK) {
exit(0);
}
fprintf (stderr,"ICE::ApplicationServer: port=%d
```

-continued

```
ServerIP=%s\n",SrvComm_getPort ( ), SrvComm_getIPAddress ( ));
fprintf (stderr,"\n\n\n\n");
fflush(stderr);
/*
* call user-defined function to initialize the application
*
* Requirements on this function:
* 1. This function initializes any data structures in the application
* that need to be initialized before messages incoming messages
* from the client get handled.
* 2. This function must return. This means that if the application has
* its own notifier loop that, say, listens to messages from a database,
* then that notifier loop must be called in a new thread.
*/
if (!initializeApplication ( )) {
exit(0);
}
/*
* Accept a connection from the client.
* (blocks until a connect attempt is made, or until the
* acceptTimeout expires).
*/
if (SrvComm acceptClientConnection ( ) != SRVCOMM_STATUS_OK) {
exit(0);
}
/*
* Finally, we start up the server loops.
* This function does not return until everything is finished.
Code Example C-2 default_main. cc Listing (Continued)
*/
SrvComm_start ( );
}
/**
* This function is called when the server times out.
* It returns TRUE or FALSE (int 1 or 0), TRUE indicating that the
* timeout-shutdown should proceed, FALSE indicating that no, don't
* shutdown on this timeout (the timeout timer gets reset).
*
*/
*/ #ifdef_cplusplus
extern "C" {
#endif
int handleServerTimeout ( ) {
return 1;
}
#ifdef_cplusplus
}
#endif
```

C Server Program Template

Code Example C-3 server-temp late. c Listing

```
/**
* server-template. c
*
*ICE-T server template.
*
*/
#include   <stdio.h>
#include   <string.h>
#include   <sys/types.h>
#include   "SrvData.h"
#include   "SrvComm.h"
#include   "SrvMessage.h"
#define false 0
#define true 1
/*This is the Message handler function that gets called when a specified *
message is received from the client. It has only one argument * (msg) that
contains the message name and message data.
*/
void handleMessage(SrvMessage *message) {
/*
```

-continued

Code Example C-3 server-temp late. c Listing

```
*disassemble incoming data
*/
/*
SrvData *data; data=SrvMessage-getDataElement (message, 0);
*/
CodeExample C-3 server-template.c Listing (Continued)
* The application "Logic" ,/
Assemble outgoing data,/
- - see SrvData.h for how to create data types. - -SrvData ,replyData; replyData=
create-SrvVector( );
SrvVector-addElement(replyData, create-SrvString("Something")); ,/
/
Create the SrvMessage struct ,/
/ reply=create-SrvMessage("replyMessage");
SrvMessage-addDataElement(replyData); ,/
/*
*Send message ,/
/*SrvComm-sendMessage(reply); ,/
/*This is an example of a shutdown handler on the server */
void shutdownHandler(SrvMessage ,shutdownMsg) {
SrvData ,shutdownTypeSStr= SrvMessage-getDataElement(shutdownMsg, 0);
SrvData *reasonSStr= SrvMessage-getDataElement(shutdownMsg, 1); char
*reasonString=SrvString-getValue(reasonSStr);
Code Example C-3 server-template. c Listing (Continued)
char *shutdownType=SrvString-getValue(shutdownTypeSStr);
if(.'strcmp(shutdownType,"ICET-CLIENT-SHUTDOWN")) {
fprintf(stderr,"shutdownHandler: Detected CLIENT SHUTDOWN event.
Reason=%s\n", reasonString);
} else if (:strcmp(shutdownType,"ICET-CLIENT-DISCONNECT")) {
fprintf(stderr,"shutdownHandler: Detected CLIENT DISCONNECT event.
Reason=%s\n", reasonString);
} else if (!strcmp(shutdownType,"ICET-SERVER-SHUTDOWN")) {
fprintf(stderr,"shutdownHandler: Detected SERVER SHUTDOWN event.
Reason=%s\n", reasonString);
} else if (:strcmp(shutdownType,"ICET-INTERNAL-SHUTDOWN")) {
fprintf(stderr,"shutdownHandler: Detected INTERNAL SHUTDOWN event.
Reason=%s\n", reasonString);
} else {
fprintf(stderr,"Shutdown Handler: unknown msg type (%s). Reason=%s\n",
shutdownType, reasonString);
}
}
*****************************************************************
*                                                               *
* Functions required by default-main, that set the              *
* SrvComm library properties (setSrvCommProperties) and         *
* initialize the applicationService (initializeApplicationService)*
*                                                               *
*****************************************************************
int setSrvCommProperties( ) {
/*The server timeout is how long the server waits between receiving
*messages from the PE before it initiates the timeout shutdown procedure.
*/SrvComm-setServerTimeout(300);/, in seconds */
/*The accept timeout is how long the server will wait to accept a connection *
from the client before it shuts down.
*/ SrvComm-setAcceptTimeout(400);/* in seconds */
return 1;
}
int createMessageHandlers( )
{ SrvComm-addMessageHandler("SampleIncomingMesssage", handleMessage);
SrvComm-addMessageHandler("ICET-SHUTDOWN", shutdownHandler);
return 1;
}
int initializeApplication( ) {
return 1;
}
```

Code Example C-4 default-main.c Listing

```
**********************************************************
default-main.c
```

This file implements the default main( ) used by an ICE-T server executable.

The structure of this main is that it calls

all the required functions necessary to run

the ICE-T SrvComm library, and provides several places for the application writer to include application-specific code. These functions are required to be implemented by the application-writer, and are usually implemented in a separate file, that gets linked in as part of the build process.

These functions all return int values, which are used to determine if the startup of the application should continue or should be aborted.

In this way the application code has a hook for indicating when something goes wrong (can't connect to database, or something), and passing this information on to the application so startup doesn't continue.

Return 1 (TRUE) to indicate that everything is OK, or return 0 (FALSE) to indicate a fatal problem that should result in aborting the application startup. int setSrvCommProperties( )

Fill in this function to set properties on the SrvComm library. This function is called after the SrvComm library is created, but before it accepts a connection from the client.

int createMessageHandlers( )

Fill in this function to register the message handlers.

This function is called immediately after setSrvCommProperties( ).

int initializeApplication( )

Fill in this function to start up the application-specific code. It is called after the connection is accepted from the client, and before the message reader and handler loops are started.

NOTES:

This main( ) routine is provided for your convenience,

CodeExample C-4 default-main.c Listing (Continued)

but you are not required to use it. You are free to write your own main( ) routine, provided you follow the requirements for ordering the calls to the SrvComm library:

1. First call create-SrvComm("tcp"); to create the Comm library's data structures.

2. Set the properties on the Comm library and register message handlers and other handlers.

3. Call SrvComm-createSocket( ) to create the listener socket and output the port number (Note: do NOT write anything to stdout before * making this call).

4. Initialize the application.

5. Call SrvComm-acceptClientConnection( ) to accept the connection from the client. (Note: steps 4 and 5 may be interchanged)

6. Finally call SrvComm-start( ) to start the Comm library's loops. This routine does not return until everything is finished.

```
**********************************************************
#include   <stdio.h>
#include   <stdlib.h>
#include   <string.h>
#include   <sys/types.h>
#include "SrvComm.h"
extern int setSrvCommProperties( );
extern int createMessageHandlers( );
extern int initializeApplication( );
*****************************************
*     The default Main                                  *
*                                                       *
*     ICE-T provides a default main( ) routine          *
*     for ICE-T server applications.                    *
*****************************************
int
main(
intargc,
char**argv,
char**envp
)
{
char ,protocol="tcp";
create-SrvComm(protocol);
/*
*Call user-defined function to set the properties on the Comm Library. * Then
call user-defined function to create the Message handlers * and add them to the
Comm library.
The setSrvCommProperties function is where Comm Library properties are set.
All properties have reasonable defaults, so none of them are required to be set.
The createMessageHandlers function is where the message handlers are added,
and other handlers (shutdown handler, default message handler, etc) are
registered.
*Requirements on these two functions:
* 1.    These functions may NOT write anything to stdout, since the Access,
```

-continued

```
        layer expects the first thing on stdout to be the port number that *
        the server will be listening for a client connection on.
*
*/if (;setSrvCommProperties( ) ) {
exit(O);
}
if (!createMessageHandlers( ) ) {
exit(0);
}
/*
* Create the socket that the client will connect to.
*This routine also is responsible for generating (and printing to
*stdout) the port number so the Access layer can generate the
*html code that passes the port number to the client applet
CodeExample C-4 default-main.c LLstmg (Continued)
* (presentation engine).
*
*(NOTE: Do not write anything to stdout before this routine is called.
*writing to stderr is OK, however.)
*/
if (SrvComm-createSocket( ) != SRVCOMM-STATUS-OK) {
exit(O);
fprintf(stderr,"ICE::ApplicationServer: port=%d
ServerIP=%s\n",SrvComm-getPort( ), SrvComm-getIPAddress( ));
fprintf(stderr,"\n\n\n\n");
fflush(stderr);
/*
* call user-defined function to initialize the application
*
*Requirements on this function:
    *  1. This function initializes any data structures in the application
    *     that need to be initialized before messages incoming messages
    *     from the client get handled.
*   2.    This function must return. This means that if the application has
          its own notifier loop that, say, listens to messages from a
          database, then that notifier loop must be called in a new thread.
*/
if (!initializeApplication( ) ) {
exit (0);
}
/,
*Accept a connection from the client.
*(blocks until a connect attempt is made, or until the
*acceptTimeout expires).
*/if (SrvComm-acceptClientConnection( ) != SRVCOMM-STATUS-OK) {
exit(O);
/*
*Finally, we start up the server loops.
*This function does not return until everything is finished.
SrvComm-start( );
}
/**
This function is called when the server times out.
It returns TRUE or FALSE (int 1 or 0), TRUE indicating that the
timeout-shutdown should proceed, FALSE indicating that no, don't
shutdown on this timeout (the timeout timer gets reset).
*/
#ifdef - -cplusplus
extern "C"
#endif
int handleServerTimeout( ) {
return 1;
}
#ifdef - -cplusplus }
#endif
```

ICE-T Exceptions Catalog D

ICE-T client program exceptions are caught by the modules in the Presentation Engine. The IcetExcelDtionHandler generates exception messages. It issues warnings and errors by printing messages to the Java console in a Netscape Navigator environment, or to the terminal where Java was started (in the case of the Java development kit).

Server (Communication Library) exceptions are caught by the modules in the Communication Layer. The server-ExceptionHandler generates messages. It issues warnings and errors by printing messages to the application logfile if the application is launched from a browser or to stdout if the application is run standAlone.

Here is a sample message:

ICET WARNING: handled in PresentationEngine.init

Attempt to register duplicate handlers.

FIG. **30** is a table of client and server side exceptions in accordance with a preferred embodiment.

While the invention is described in terms of preferred embodiments in a specific system environment, those skilled

in the art will recognize that the invention can be practiced, with modification, in other and different hardware and software environments within the spirit and scope of the appended claims.

Having thus described our invention, what we claim as new, and desire to secure by Letters Patent is:

1. A server for a distributed system, comprising:

(a) a plurality of client computers;

(b) a plurality of server computers;

(c) a network connecting the plurality of client computers to the plurality of server computers;

(d) a client computer which executes a client computer application which gathers information about the client computer and contacts a server computer using the network;

(e) the server computer authenticates the information about the client computer and initiates a server computer code segment responsible for communicating with the client computer; and

(f) the server computer transmits a client computer code segment for execution at the client computer to facilitate communication between the client computer and the server computer.

2. The server for a distributed system as recited in claim 1, including a plurality of definitions that define the plurality of client computer code segments and each of the plurality of definitions defining how to associate the plurality of client computer code segments and the plurality of server computer code segments into applications in response to a request by the client computer.

3. The server for a distributed system as recited in claim 1, in which the server computer code segment includes at least one instance of a communication library code segment configured to couple an associated one of the plurality of server computer code segments to the execution framework code segment, and the client computer includes at least one instance of a communication library code segment configured to couple an associated one of the plurality of client code segments transmitted to the client computer to the execution framework code segment.

4. The server for a distributed system as recited in claim 1, including a web connection which facilitates communication from the client computer to the server computer.

5. The server for a distributed system as recited in claim 4, wherein the web is the Internet.

6. The server for a distributed system as recited in claim 1, including a communication protocol corresponding to the client code segment and the server code segment which is supported by a communication library on the client computer and the server computer.

7. The server for a distributed system as recited in claim 1, including a code segment which authenticates an initial request from a client computer before downloading a client code segment and initiating secure communication.

8. The server for a distributed system as recited in claim 1, wherein a listener socket is opened on the server, and the port number of the listener socket is transmitted as part of the code segment transmitted to the client computer.

9. The server for a distributed system as recited in claim 1, wherein authentication information includes a password.

10. A method for enabling a distributed system, including a plurality of client computers coupled via a network to a plurality of server computers, comprising the steps of:

(a) executing a client computer application on the client computer which gathers information about the client computer and contacts a server computer using the network;

(b) authenticating the information about the client computer at the server computer;

(c) initiating a server computer code segment to communicate with the client computer; and

(d) transmitting a client computer code segment for execution at the client computer to facilitate communication between the client computer and the server computer.

11. The method as recited in claim 10, including invoking a plurality of definitions that define the plurality of client computer code segments and each of the plurality of definitions defining how to associate the plurality of client computer code segments and the plurality of server computer code segments into applications in response to a request by the client computer.

12. The method as recited in claim 11, in which the server computer code segment includes at least one instance of a communication library code segment configured to couple an associated one of the plurality of server computer code segments to the execution framework code segment, and the client computer includes at least one instance of a communication library code segment configured to couple an associated one of the plurality of client code segments transmitted to the client computer to the execution framework code segment.

13. The method as recited in claim 10, including the step of utilizing a web connection which facilitates communication from the client computer to the server computer.

14. The method as recited in claim 13, wherein the web is the Internet.

15. The method as recited in claim 10, including the step of using a communication protocol corresponding to the client code segment and the server code segment which is supported by a communication library on the client computer and the server computer.

16. The method as recited in claim 10, including the step of authenticating an initial request from a client computer before downloading a client code segment and initiating secure communication.

17. The method as recited in claim 10, wherein a listener socket is opened on the server, and the port number of the listener socket is transmitted as part of the code segment transmitted to the client computer.

18. The method as recited in claim 10, wherein authentication information includes a password.

19. A computer program embodied on a computer-readable medium for enabling a distributed computer system, comprising:

(a) a client computer code segment for residence on a client computer and including an user interface;

(b) a server computer code segment for residence on a server computer coupled to the client computer;

(c) a code segment to enable a connection to a network connecting a plurality of client computers to a plurality of server computers;

(d) a code segment which executes a client computer application on the client computer which gathers information about the client computer and contacts the server computer using the network;

(e) a code segment which authenticates the information about the client computer and initiates the server computer code segment responsible for communicating with the client computer; and

(f) a code segment which transmits the client computer code segment for execution at the client computer to facilitate communication between the client computer and the server computer.

20. The computer program embodied on a computer-readable medium for enabling a distributed computer system as recited in claim 19, including a plurality of definitions that define the plurality of client computer code segments and each of the plurality of definitions defining how to associate the plurality of client computer code segments and the plurality of server computer code segments into applications in response to a request by the client computer.

21. The computer program embodied on a computer-readable medium for enabling a distributed computer system as recited in claim 19, in which the server computer code segment includes at least one instance of a communication library code segment configured to couple an associated one of the plurality of server computer code segments to the execution framework code segment, and the client computer includes at least one instance of a communication library code segment configured to couple an associated one of the plurality of client code segments transmitted to the client computer to the execution framework code segment.

22. The computer program embodied on a computer-readable medium for enabling a distributed computer system as recited in claim 19, including a web connection which facilitates communication from the client computer to the server computer.

23. The computer program embodied on a computer-readable medium for enabling a distributed computer system as recited in claim 22, wherein the web is the Internet.

24. The computer program embodied on a computer-readable medium for enabling a distributed computer system as recited in claim 19, including a communication protocol corresponding to the client code segment and the server code segment which is supported by a communication library on the client computer and the server computer.

25. The computer program embodied on a computer-readable medium for enabling a distributed computer system as recited in claim 19, including a code segment which authenticates an initial request from a client computer before downloading a client code segment and initiating secure communication.

26. The computer program embodied on a computer-readable medium for enabling a distributed computer system as recited in claim 19, wherein a listener socket is opened on the server, and the port number of the listener socket is transmitted as part of the code segment transmitted to the client computer.

27. The computer program embodied on a computer-readable medium for enabling a distributed computer system as recited in claim 19, wherein authentication information includes a password.

* * * * *